Text Analytics with Python

A Practitioner's Guide to Natural Language Processing

—

Second Edition

—

Dipanjan Sarkar

Apress®

Text Analytics with Python

A Practitioner's Guide to Natural Language Processing

Second Edition

Dipanjan Sarkar

Apress®

Text Analytics with Python

Dipanjan Sarkar
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-4353-4 ISBN-13 (electronic): 978-1-4842-4354-1
https://doi.org/10.1007/978-1-4842-4354-1

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4353-4. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my dear readers, partner,
pets, friends, family, and well-wishers.

—Dipanjan Sarkar

Table of Contents

About the Author xv

About the Technical Reviewer xvii

Foreword xix

Acknowledgments xxi

Introduction xxiii

Chapter 1: Natural Language Processing Basics 1

Natural Language 3

What Is Natural Language? 3

The Philosophy of Language 3

Language Acquisition and Usage 6

Linguistics 10

Language Syntax and Structure 13

Words 15

Phrases 17

Clauses 20

Grammar 21

Word-Order Typology 33

Language Semantics 35

Lexical Semantic Relations 35

Semantic Networks and Models 39

Representation of Semantics 41

Text Corpora 51

Corpora Annotation and Utilities 52

Popular Corpora 53

Accessing Text Corpora 55

Natural Language Processing 62
- Machine Translation 62
- Speech Recognition Systems 63
- Question Answering Systems 64
- Contextual Recognition and Resolution 64
- Text Summarization 65
- Text Categorization 65

Text Analytics 66

Machine Learning 67

Deep Learning 68

Summary 68

Chapter 2: Python for Natural Language Processing 69

Getting to Know Python 70

The Zen of Python 71

Applications: When Should You Use Python? 73

Drawbacks: When Should You Not Use Python? 75

Python Implementations and Versions 76

Setting Up a Robust Python Environment 78
- Which Python Version? 78
- Which Operating System? 79
- Integrated Development Environments 79
- Environment Setup 80
- Package Management 84
- Virtual Environments 85

Python Syntax and Structure 88

Working with Text Data 89
- String Literals 89
- Representing Strings 91
- String Operations and Methods 93

Basic Text Processing and Analysis: Putting It All Together 106

Natural Language Processing Frameworks 111

Summary... 113

Chapter 3: Processing and Understanding Text.. 115

Text Preprocessing and Wrangling... 117

Removing HTML Tags 117

Text Tokenization 119

Removing Accented Characters... 135

Expanding Contractions.. 136

Removing Special Characters.. 138

Case Conversions 138

Text Correction... 139

Stemming 148

Lemmatization.. 152

Removing Stopwords.. 154

Bringing It All Together—Building a Text Normalizer 155

Understanding Text Syntax and Structure.. 157

Installing Necessary Dependencies.. 159

Important Machine Learning Concepts... 162

Parts of Speech Tagging... 163

Shallow Parsing or Chunking 172

Dependency Parsing... 183

Constituency Parsing.. 190

Summary.. 199

Chapter 4: Feature Engineering for Text Representation.................................... 201

Understanding Text Data... 202

Building a Text Corpus 203

Preprocessing Our Text Corpus... 205

Traditional Feature Engineering Models... 208

Bag of Words Model.. 208

Bag of N-Grams Model 210
TF-IDF Model 211
Extracting Features for New Documents 220
Document Similarity 220
Topic Models 226
Advanced Feature Engineering Models 231
Loading the Bible Corpus 233
Word2Vec Model 234
Robust Word2Vec Models with Gensim 255
Applying Word2Vec Features for Machine Learning Tasks 258
The GloVe Model 263
Applying *GloVe* Features for Machine Learning Tasks 265
The FastText Model 269
Applying FastText Features to Machine Learning Tasks 270
Summary 273
Chapter 5: Text Classification 275
What Is Text Classification? 277
Formal Definition 277
Major Text Classification Variants 278
Automated Text Classification 279
Formal Definition 281
Text Classification Task Variants 282
Text Classification Blueprint 282
Data Retrieval 285
Data Preprocessing and Normalization 287
Building Train and Test Datasets 292
Feature Engineering Techniques 293
Traditional Feature Engineering Models 294
Advanced Feature Engineering Models 295

Classification Models 296
Multinomial Naïve Bayes 298
Logistic Regression 301
Support Vector Machines........ 303
Ensemble Models 306
Random Forest 307
Gradient Boosting Machines........ 308
Evaluating Classification Models 309
Confusion Matrix 310
Building and Evaluating Our Text Classifier........ 315
Bag of Words Features with Classification Models........ 315
TF-IDF Features with Classification Models 319
Comparative Model Performance Evaluation 322
Word2Vec Embeddings with Classification Models........ 323
GloVe Embeddings with Classification Models........ 326
FastText Embeddings with Classification Models........ 327
Model Tuning 328
Model Performance Evaluation........ 334
Applications 341
Summary........ 341
Chapter 6: Text Summarization and Topic Models 343
Text Summarization and Information Extraction........ 344
Keyphrase Extraction........ 346
Topic Modeling 346
Automated Document Summarization........ 346
Important Concepts........ 347
Keyphrase Extraction 350
Collocations 351
Weighted Tag-Based Phrase Extraction........ 357

Topic Modeling 362
Topic Modeling on Research Papers 364
The Main Objective 364
Data Retrieval 365
Load and View Dataset 366
Basic Text Wrangling 367
Topic Models with Gensim 368
Text Representation with Feature Engineering 369
Latent Semantic Indexing 372
Implementing LSI Topic Models from Scratch 382
Latent Dirichlet Allocation 389
LDA Models with MALLET 399
LDA Tuning: Finding the Optimal Number of Topics 402
Interpreting Topic Model Results 409
Predicting Topics for New Research Papers 415
Topic Models with Scikit-Learn 418
Text Representation with Feature Engineering 419
Latent Semantic Indexing 419
Latent Dirichlet Allocation 425
Non-Negative Matrix Factorization 428
Predicting Topics for New Research Papers 432
Visualizing Topic Models 434
Automated Document Summarization 435
Text Wrangling 439
Text Representation with Feature Engineering 440
Latent Semantic Analysis 441
TextRank 445
Summary 450

Chapter 7: Text Similarity and Clustering 453
Essential Concepts 455
Information Retrieval (IR) 455
Feature Engineering 455
Similarity Measures 456
Unsupervised Machine Learning Algorithms 457
Text Similarity 457
Analyzing Term Similarity 458
Hamming Distance 461
Manhattan Distance 462
Euclidean Distance 464
Levenshtein Edit Distance 465
Cosine Distance and Similarity 471
Analyzing Document Similarity 475
Building a Movie Recommender 476
Load and View Dataset 477
Text Preprocessing 480
Extract TF-IDF Features 481
Cosine Similarity for Pairwise Document Similarity 482
Find Top Similar Movies for a Sample Movie 483
Build a Movie Recommender 484
Get a List of Popular Movies 485
Okapi BM25 Ranking for Pairwise Document Similarity 488
Document Clustering 497
Clustering Movies 500
Feature Engineering 500
K-Means Clustering 501
Affinity Propagation 508
Ward's Agglomerative Hierarchical Clustering 512
Summary 517

Chapter 8: Semantic Analysis 519

- Semantic Analysis 520
- Exploring WordNet 521
  - Understanding Synsets 522
  - Analyzing Lexical Semantic Relationships 523
- Word Sense Disambiguation 533
- Named Entity Recognition 536
- Building an NER Tagger from Scratch 544
- Building an End-to-End NER Tagger with Our Trained NER Model 554
- Analyzing Semantic Representations 558
  - Propositional Logic 558
  - First Order Logic 560
- Summary 566

Chapter 9: Sentiment Analysis 567

- Problem Statement 568
- Setting Up Dependencies 569
- Getting the Data 569
- Text Preprocessing and Normalization 570
- Unsupervised Lexicon-Based Models 572
  - Bing Liu’s Lexicon 574
  - MPQA Subjectivity Lexicon 574
  - Pattern Lexicon 575
  - TextBlob Lexicon 575
  - AFINN Lexicon 578
  - SentiWordNet Lexicon 580
  - VADER Lexicon 584
- Classifying Sentiment with Supervised Learning 587
- Traditional Supervised Machine Learning Models 590
- Newer Supervised Deep Learning Models 593
- Advanced Supervised Deep Learning Models 602

Analyzing Sentiment Causation 614
- Interpreting Predictive Models 614
- Analyzing Topic Models 622

Summary 629

Chapter 10: The Promise of Deep Learning 631

Why Are We Crazy for Embeddings? 633

Trends in Word-Embedding Models 635

Trends in Universal Sentence-Embedding Models 636

Understanding Our Text Classification Problem 642

Universal Sentence Embeddings in Action 643
- Load Up Dependencies 643
- Load and View the Dataset 644
- Building Train, Validation, and Test Datasets 645
- Basic Text Wrangling 645
- Build Data Ingestion Functions 647
- Build Deep Learning Model with Universal Sentence Encoder 648
- Model Training 649
- Model Evaluation 651

Bonus: Transfer Learning with Different Universal Sentence Embeddings 652

Summary and Future Scope 659

Index 661

About the Author

Dipanjan Sarkar is a Data Scientist at Red Hat, a published author, and a consultant and trainer. He has consulted and worked with several startups as well as Fortune 500 companies like Intel. He primarily works on leveraging data science, advanced analytics, machine learning, and deep learning to build large-scale intelligent systems. He holds a master of technology degree with specializations in data science and software engineering. He is also an avid supporter of self-learning and massive open online courses. He has recently ventured into the world of open source products to improve the productivity of developers across the world.

Dipanjan has been an analytics practitioner for several years now, specializing in machine learning, natural language processing, statistical methods, and deep learning. Having a passion for data science and education, he also acts as an AI consultant and mentor at various organizations like Springboard, where he helps people build their skills on areas like data science and machine learning. He also acts as a key contributor and editor for *Towards Data Science,* a leading online journal focusing on artificial intelligence and data science. Dipanjan has also authored several books on R, Python, machine learning, social media analytics, natural language processing, and deep learning.

Dipanjan's interests include learning about new technology, financial markets, disruptive start-ups, data science, artificial intelligence, and deep learning. In his spare time, he loves reading, gaming, watching popular sitcoms and football, and writing interesting articles on https://medium.com/@dipanzan.sarkar and https://www.linkedin.com/in/dipanzan. He is also a strong supporter of open source and publishes his code and analyses from his books and articles on GitHub at https://github.com/dipanjanS.

About the Technical Reviewer

Santanu Pattanayak currently works at GE, Digital as a Staff Data Scientist and is the author of the deep learning book P*ro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python*. He has around 12 years of overall work experience with eight of years of experience in the data analytics/data science field. He also has a background in development and database technologies. Prior to joining GE, Santanu worked in companies such as RBS, Capgemini, and IBM. He graduated with a degree in electrical engineering from Jadavpur University, Kolkata and is an avid math enthusiast. Santanu is currently pursuing a master's degree in data science from Indian Institute of Technology (IIT), Hyderabad. He also devotes his time to data science hackathons and Kaggle competitions where he ranks within the top 500 across the globe. Santanu was born and brought up in West Bengal, India and currently resides in Bangalore, India with his wife.

Foreword

The power of text analytics and natural language processing is beginning to live up to its promise, thanks to contemporary developments in machine learning.

If you have read Dipanjan Sarkar's *Text Analytics with Python: A Practical Real-World Approach to Gaining Actionable Insights from your Data*, then you probably already have some sense that this is true. Released in 2016, this book has quickly become a staple in the natural language processing community. Yet, in the world of technology, 2 years can seem like a lifetime, and so welcome to the updated second edition of *Text Analytics with Python*!

While the core of the first edition's original material has been preserved, there are a number of updates and changes throughout. Of note, text classification and sentiment analysis have been expanded to include both traditional machine learning and deep learning models, important as neural networks become increasingly central in approaches to natural language processing. Additionally, topic modeling, a collection of techniques for abstract topic discovery, has been further developed to include a number of complementary methods, and to leverage additional Python libraries.

There is also an entire new chapter on feature engineering – which plays an especially central role in natural language processing and text data – where both traditional and neural network-based methods are covered. In addition, as much as deep learning is discussed in terms of natural language processing these days, there is a palpable sense that it is only the beginning; to that end, an entire new chapter is dedicated to the promise of deep learning for natural language processing.

Why *Text Analytics with Python*? Not only does this book cover the ideas and intuitions behind various cutting-edge text analytics and natural language processing tasks, it thoroughly presents practical approaches and Python code to cement these ideas, in order for the reader to put them to use for themselves. Since Sarkar has already proven the worth of his knowledge and instruction on text analytics, having a look at the second edition, expanded and updated throughout, can be classified as a great idea.

—Matthew Mayo
Editor, KDnuggets
@mattmayo13

Acknowledgments

This book would have definitely not been a reality without the help and support from some excellent people and organizations that have helped us along this journey. First and foremost, a big thank you to all our readers for not only reading our books but also supporting us with valuable feedback and insights. Truly, I have learned a lot from all of you and still continue to do so. You have helped us make the new edition of this book a reality with your feedback! I would also like to acknowledge the entire team at Apress for working tirelessly behind the scenes to create and publish quality content for everyone.

A big shout-out goes to the entire Python developer community, especially to the developers of frameworks like NumPy, SciPy, Scikit-Learn, spaCy, NLTK, Pandas, Gensim, Keras, TextBlob, and TensorFlow. Kudos to organizations like Anaconda, for making the lives of data scientists easier and for fostering an amazing ecosystem around data science, artificial intelligence, and natural language processing that has been growing exponentially with time.

I also thank my friends, colleagues, teachers, managers, and well-wishers for supporting me with excellent challenges, strong motivation, and good thoughts. A lot of the content in this book wouldn't have been possible without the help from several people and some excellent resources. We would like to thank Christopher Olah, for providing some excellent depictions and explanations for LSTM models (http://colah.github.io); Pramit Choudhary, for helping us cover a lot of ground in model interpretation with Skater; François Chollet, for creating Keras and writing an excellent book on deep learning; Raghav Bali, who has co-authored several books with me and helped me reframe a lot of the content from this book; Srdjan Santic, for being an excellent spokesman of this book and giving me a lot of valuable feedback; Matthew Mayo, for being so kind in gracing us with writing the foreword for this book and publishing amazing content on KDnuggets; and my entire team at Towards Data Science, Springboard, and Red Hat for helping me learn and grow every day. Also thanks to industry experts, including Kirk Borne, Tarry Singh, Favio Vazquez, Dat Tran, Matt Dancho, Kate Strachnyi, Kristen Kehrer, Kunal Jain, Sudalai Rajkumar, Beau Walker, David Langer, Andreas Kretz and many others for helping me learn more everyday and for keeping me motivated.

I would also like to acknowledge and express my gratitude to my parents, Digbijoy and Sampa, my partner Durba, my pets, family, and well-wishers for their constant love, support, and encouragement, which drives me to strive to achieve more. Finally, once again I would like to thank the entire team at Apress, especially Welmoed Spahr, Aditee Mirashi, Celestin John, our editors, and Santanu Pattanayak for being a part of this wonderful journey.

—Dipanjan Sarkar

Introduction

Data is the new oil and unstructured data—especially text, images, and videos—contains a wealth of information. However, due to the inherent complexity in processing and analyzing this data, people often refrain from spending extra time and effort venturing out from structured datasets to analyze these unstructured sources of data, which can be a potential gold mine. Natural language processing (NLP) is all about leveraging tools, techniques, and algorithms to process and understand natural language-based data, which is usually unstructured like text, speech, and so on. In this book, we will be looking at tried and tested strategies—techniques and workflows—that can be leveraged by practitioners and data scientists to extract useful insights from text data.

Being specialized in domains like computer vision and natural language processing is no longer a luxury but a necessity expected of any data scientist in today's fast-paced world! *Text Analytics with Python* is a practitioner's guide to learning and applying NLP techniques to extract actionable insights from noisy and unstructured text data. This book helps its readers understand essential concepts in NLP along with extensive case studies and hands-on examples to master state-of-the-art tools, techniques, and frameworks for actually applying NLP to solve real-world problems. We leverage Python 3 and the latest and best state-of-the-art frameworks, including NLTK, Gensim, spaCy, Scikit-Learn, TextBlob, Keras, and TensorFlow, to showcase the examples in the book. You can find all the examples used in the book on GitHub at https://github.com/dipanjanS/text-analytics-with-python.

In my journey in this field so far, I have struggled with various problems, faced many challenges, and learned various lessons over time. This book contains a major chunk of the knowledge I've gained in the world of text analytics and natural language processing, where building a fancy word cloud from a bunch of text documents is not enough anymore. Perhaps the biggest problem with regard to learning text analytics is not a lack of information but too much information, often called *information overload*. There are so many resources, documentation, papers, books, and journals containing so much content that they often overwhelm someone new to the field. You might have had questions like, "What is the right technique to solve a problem?," "How does text summarization really work?," and "Which frameworks are best for solving multi-class text categorization?," among many others! By combining mathematical and theoretical concepts with practical implementations of real-world case studies using Python, this book tries to address this problem and help readers avoid the pressing issues I've faced in my journey so far.

This book follows a comprehensive and structured approach. First it tackles the basics of natural language understanding and Python for handling text data in the initial chapters. Once you're familiar with the basics, we cover text processing, parsing, and understanding. Then, we address interesting problems in text analytics in each of the remaining chapters, including text classification, clustering and similarity analysis, text summarization and topic models, semantic analysis and named entity recognition, and sentiment analysis and model interpretation. The last chapter is an interesting chapter on the recent advancements made in NLP thanks to deep learning and transfer learning and we cover an example of text classification with universal sentence embeddings.

The idea of this book is to give you a flavor of the vast landscape of text analytics and NLP and to arm you with the necessary tools, techniques, and knowledge to tackle your own problems. I hope you find this book helpful and wish you the very best in your journey through the world of text analytics and NLP!

CHAPTER 1

Natural Language Processing Basics

We have ushered in the age of Big Data, where organizations and businesses are having difficulty managing all the data generated by various systems, processes, and transactions. However, the term Big Data is misused a lot due to the vague definition of the 3Vs of data—volume, variety, and velocity. It is sometimes difficult to quantify what data is "big". Some might think a billion records in a database is "Big Data," but that number seems small compared to the petabytes of data being generated by various sensors or by social media. One common characteristic is the large volume of unstructured textual data that's present across all organizations, irrespective of their domain. As an example, we have vast amounts of data in the form of tweets, status messages, hash tags, articles, blogs, wikis, and much more on social media. Even retail and ecommerce stores generate a lot of textual data, from new product information and metadata to customer reviews and feedback.

The main challenges associated with textual data are two-fold. The first challenge deals with effective storage and management of this data. Textual data is usually unstructured and does not adhere to any specific predefined data model or schema followed by relational databases. However, based on the data semantics, you can store it in SQL-based database management systems like SQL Server and MySQL, in NoSQL-based systems like MongoDB and CouchDB, and more recently in information retrieval-based data stores like ElasticSearch and Solr.

Organizations with enormous amounts of textual datasets often resort to warehouses and file-based systems like Hadoop, where they dump all the data in the Hadoop Distributed File System (HDFS) and access it as needed. This is one of the main principles of a data lake.

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_1

The second challenge is with regard to analyzing this data and trying to extract meaningful patterns and actionable insights from it. Even though we have a large number of machine learning and data analysis techniques at our disposal, the majority of them are tuned to work with numerical data. Hence, we have to resort to natural language processing and specialized techniques and transformations and models to analyze text data or more specifically natural language. This is quite different from structured data and normal programming languages, which are easily understood by machines. Remember that textual data is highly unstructured, so it does not adhere to structured or regular syntax and patterns. Hence, we cannot directly use statistical or machine learning models to analyze such data.

Unstructured data—especially text, images, and videos—contain a wealth of information. However, due to the inherent complexity in processing and analyzing this data, people often refrain from venturing out from structured datasets to analyze these unstructured sources of data, which can be a potential gold mine. Natural language processing (NLP) is all about leveraging tools, techniques, and algorithms to process and understand natural language-based data, which is usually unstructured (like text, speech, and so on). We cover essential concepts and techniques around NLP in this book. However, before we dive into specific techniques or algorithms to analyze textual data, we cover some of the core concepts and principles associated with natural language and unstructured text. The primary intent of this is to familiarize you with concepts and domains associated with natural language processing and text analytics.

We use the Python programming language in this book primarily for accessing and analyzing textual data. Being a revised edition, we focus on Python 3.x and the latest state-of-the-art open source frameworks for our analyses. The examples in this chapter will be pretty straightforward and fairly easy to follow. However, you can quickly skim over Chapter 2, "Python for Natural Language Processing" if you want to get a head start on Python, essential frameworks, and constructs before going through this chapter.

In this chapter, we cover concepts relevant to natural language, linguistics, text data formats, syntax, semantics, and grammar (all of which are major components of NLP itself) before moving on to more advanced topics like text corpora, natural language processing, deep learning, and text analytics. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Natural Language

Textual data is unstructured data but it usually belongs to a specific language following specific syntax and semantics. All text data, such as a simple word, sentence, or a document, relates back to some natural language. In this section, we look at the definition of natural language, the philosophy of language, language acquisition, and the use of language.

What Is Natural Language?

To understand text analytics and natural language processing, we need to understand what makes a language *"natural"*. In simple terms, a natural language is a language developed and evolved by humans through natural use and communication rather than constructing and creating the language artificially, like a computer programming language.

Various human languages, such as English, Japanese, or Sanskrit, can be called natural languages. Natural languages can be communicated in different ways, including speech, writing, or even using signs. There has been a lot of interest in trying to understand the origins, nature, and philosophy of language. We discuss this briefly in the following section.

The Philosophy of Language

We now know what a natural language is. But think about the following questions. What are the origins of a language? What makes the English language "English"? How did the meaning of the word *"fruit"* come into existence? How do humans communicate using language? These are definitely some heavy philosophical questions.

We now look at the philosophy of language, which mainly deals with the following four problems.

- The nature of meaning in a language
- The use of language
- Language cognition
- The relationship between language and reality

The nature of meaning in a language is concerned with the semantics of a language and the nature of meaning itself. Here, philosophers of language or linguistics try to find out what it means to actually "mean" anything, i.e., how the meaning of any word or sentence came into being and how different words in a language can be synonyms of each other and form relations. Another thing of importance here is how structure and syntax in the language paved the way for semantics or, to be more specific, how words that have their own meaning are structured together to form meaningful sentences. *Linguistics* is the scientific study of language, a special field that deals with some of these problems.

Syntax, semantics, grammar, and parse trees are some ways to solve these problems. The nature of meaning can be expressed in linguistics between two human beings, notably a sender and a receiver. From a non-linguistic standpoint, things like body language, prior experiences, and psychological effects are contributors to the meaning of language, where each human being perceives or infers meaning in their own way, taking into account some of these factors.

The use of language is more concerned with how language is used as an entity in various scenarios and communication between human beings. This includes analyzing speech and the usage of language when speaking, including the speaker's intent, tone, content, and actions involved in expressing a message. This is often called a "speech act" in linguistics. More advanced concepts like language creation and human cognitive activities like language acquisition—which study the learning and usage of languages—are also of prime interest.

Language cognition specifically focuses on how the cognitive functions of the human brain are responsible for understanding and interpreting language. Considering the example of a typical sender and receiver, there are many actions involved, from message communication to interpretation. Cognition tries to find out how the mind combines and relates specific words into sentences and then into a meaningful message and what is the relation of language is to the thought process of the sender and receiver when they use the language to communicate messages.

The relationship between language and reality explores the extent of truth of expressions originating from language. Language philosophers try to measure how factual these expressions are and how they relate to certain affairs in our world which are true. This relationship can be expressed in several ways and we explore some of them.

One of the most popular models is the "triangle of reference," which is used to explain how words convey meaning and ideas in the minds of the receiver and how that meaning relates back to a real-world entity or fact. The triangle of reference was proposed by Charles Ogden and Ivor Richards in their book, *The Meaning of Meaning,* and is denoted in Figure 1-1.

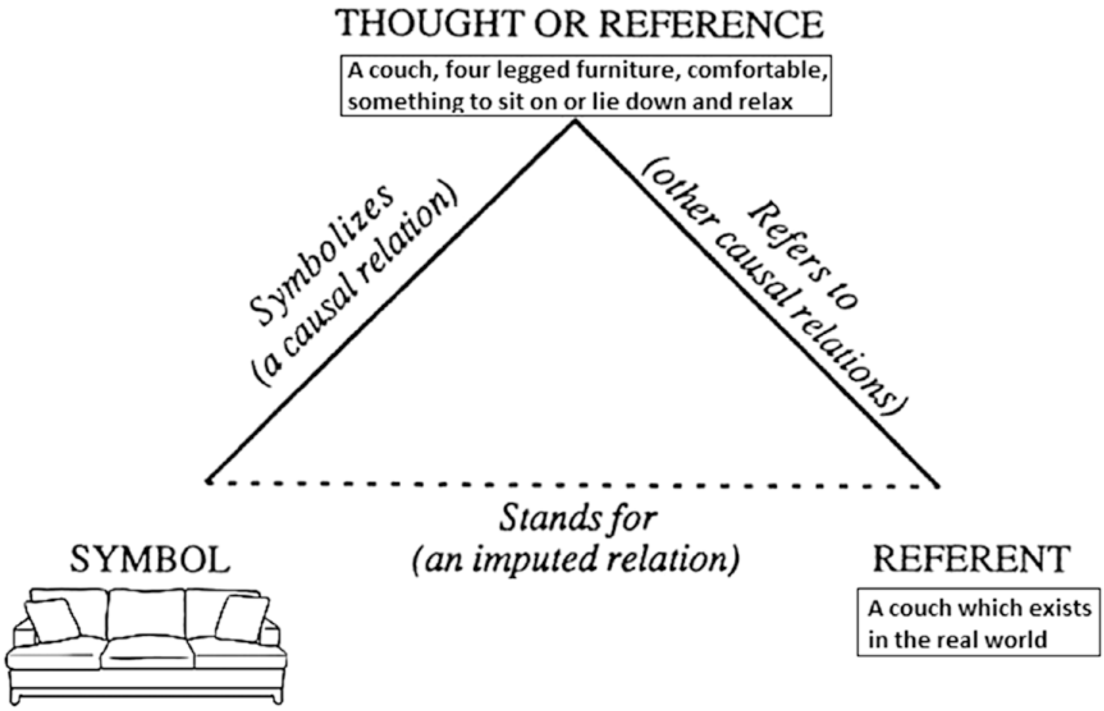

THOUGHT OR REFERENCE
A couch, four legged furniture, comfortable, something to sit on or lie down and relax
Symbolizes (a causal relation)
Refers to (other causal relations)
Stands for (an imputed relation)
SYMBOL
REFERENT
A couch which exists in the real world

Figure 1-1. *The triangle of reference model*

The triangle of reference model is also known as the meaning of meaning model. Figure 1-1 shows a real example of a couch being perceived by a person. A symbol is denoted as a linguistic symbol like a word or an object that evokes thought in a person's mind. In this case, the symbol is the couch and this evokes thoughts like what is a couch, a piece of furniture that can be used for sitting on or lying down and relaxing, something that gives us comfort. These thoughts are known as a reference and through this reference, the person is able to relate it to something that exists in the real world, which is called a referent. In this case, the referent is the couch that the person perceives to be present in front of him.

The second way to determine relationships between language and reality is known as the "direction of fit" and we talk about two main directions here. The "word-to-world" direction of fit talks about instances, where the usage of language can reflect reality. This indicates using words to match or relate to something that's happening or has already happened in the real world. An example would be the sentence, "The Eiffel Tower is really big," which accentuates a fact in reality. The other direction of fit is known as "world-to-word" and talks about instances where the usage of language can change reality. An example here would be the sentence, "I am going to take a swim," where you are changing reality by taking a swim and are representing this fact in the sentence you are communicating. Figure 1-2 shows the relationship between both directions of fits.

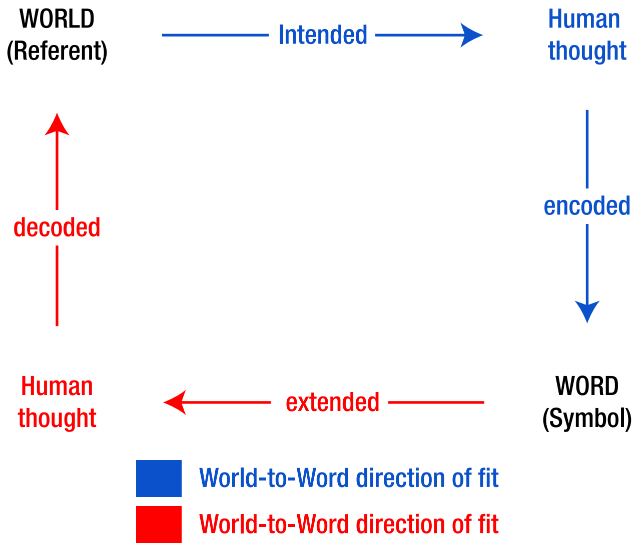


***Figure 1-2.** The direction of fit representation*

Based on the referent that is perceived from the real world, a person can form a representation in the form of a symbol or word and consequently can communicate the same to another person. This forms a representation of the real world based on the received symbol, thus forming a cycle.

Language Acquisition and Usage

By now, we have seen what natural languages mean and the concepts behind language, its nature, meaning. and use. In this section, we talk in further detail about how language is perceived, understood, and learned using cognitive abilities by humans. Finally, we end our discussion with the main forms of language usage that we discussed previously in brief, as speech acts. It is important to not only understand what natural language denotes but also how humans interpret, learn, and use language. This helps us emulate some of these concepts programmatically in our algorithms and techniques when we try to extract insights from textual data. A lot of you might have seen recent advancements based on these principles, including deep learning, sequence modeling, generative models, and cognitive computing.

Language Acquisition and Cognitive Learning

Language acquisition is defined as the process by which human beings utilize their cognitive abilities, knowledge, and experience to understand language based on hearing and perception. This enables them to start using it in terms of words, phrases, and sentences to communicate with other human beings. In simple terms, the ability of acquiring and producing languages is termed *language acquisition*.

The history of language acquisition dates back centuries, when philosophers and scholars tried to reason and understand the origins of language acquisition and came up with several theories, like it being a god-gifted ability being passed down from generation to generation. There were also scholars like Plato who indicated that a form of word-meaning mapping would have been responsible for language acquisition. Modern theories were proposed by various scholars and philosophers and some of the popular ones, most notably Burrhus Skinner, indicated that knowledge, learning, and use of language were more behavioral in nature

Symbols in any language are based on certain stimuli and are reinforced in young children's memories, based on repeated reactions to their usage. This theory is based on operant or instrumentation conditioning, which is a type of conditional learning where the strength of a particular behavior or action is modified based on its consequences like reward or punishment and these consequent stimuli help reinforce or control behavior and learning.

An example would be that a child would learn that a specific combination of sounds made up a word from repeated usage of it by his/her parents or being rewarded by appreciation when he/she speaks it correctly or being corrected when he/she makes a mistake when speaking the same. This repeated conditioning ends up reinforcing the actual meaning and understanding of the word in the child's memory. To sum it up, children try to learn and use language mostly behaviorally, by hearing and imitating adults.

However, this behavioral theory was challenged by renowned linguist Noam Chomsky, who proclaimed that it would be impossible for children to learn language just by imitating everything from adults. This hypothesis is valid in the following examples. While words like "go" and "give" are valid, children often end up using an invalid form of the word like "goed" or "gived" instead of "went" or "gave" in the past tense.

We know that their parents didn't utter these words in front of them, so it would be impossible to pick these up based on the previous theory of Skinner. Consequently, Chomsky proposed that children must not only be imitating words they hear, but also are extracting patterns, syntax, and rules from the same language constructs, and this process is separate from just utilizing generic cognitive abilities based on behavior.

Considering Chomsky's view, cognitive abilities along with language specific knowledge and abilities like syntax, semantics, parts of speech, and grammar form what he termed a "language acquisition device". This enabled humans to have the ability of "language acquisition". Besides cognitive abilities, what is unique and important in language learning is the syntax of the language itself, which can be emphasized in his famous sentence, "Colorless green ideas sleep furiously". The sentence does not make sense because colorless cannot be associated with green and neither can ideas be associated with green nor can they sleep furiously. However, the sentence is grammatically correct.

This is precisely what Chomsky tried to explain, that syntax and grammar depicts information that is independent from the meaning and semantics of words. Hence, he proposed that learning and identifying language syntax is a separate human capability compared to other cognitive abilities. This proposed hypothesis is also known as the "autonomy of syntax". Of course, these theories are still widely debated among scholars and linguists, but it is useful to consider how the human mind tends to acquire and learn language. We now look at the typical patterns in which language is generally used.

Language Usage

In the previous section, we discussed speech acts and how the direction-of-fit model is used for relating words and symbols to reality. In this section, we cover some concepts related to speech acts that highlight different ways in which language is used in communication. There are mainly three categories of speech acts. These include *locutionary*, *illocutionary,* and *perlocutionary* acts.

- Locutionary acts are mainly concerned with the actual delivery of the sentence when communicated from one human being to another by speaking it.
- *Illocutionary acts* focus on the actual semantics and significance of the sentence that was communicated.
- *Perlocutionary acts* refer to the effect the communication had on its receiver, which is more psychological or behavioral.

A simple example would be the phrase, “Get me the book from the table” spoken by a father to his child. The phrase when spoken by the father forms the locutionary act. This significance of this sentence is a directive that tells the child to get the book from the table and forms an illocutionary act. The action the child takes after hearing this, i.e. if he brings the book from the table to his father, forms the perlocutionary act. We did talk about the illocutionary act being a directive in this case. According to the philosopher John Searle, there are a total of five classes of illocutionary speech acts:

- Assertives
- Directives
- Commissives
- Expressives
- Declarations

Assertives are speech acts that communicate how things are already existent in the world. They are spoken by the sender when he tried to assert a proposition that can be *true* or *false* in the real world. These assertions could be statements or declarations. A simple example would be “The Earth revolves round the Sun”. These messages represent the word-to-world direction of fit, which we discussed earlier.

Directives are speech acts that the sender communicates to the receiver, asking or directing him/her to do something. This represents a voluntary act that the receiver might do in the future after receiving a directive from the sender. Directives can either be complied with or not complied with. These directives could be simple requests or even orders or commands. An example directive would be, “Get me the book from the table,” which we discussed in detail when we talked about types of speech acts.

Commissives are speech acts that commit the sender or speaker who utters the sentence to some future voluntary act or action. Acts like promises, oaths, pledges, and vows represent commissives and the direction of fit could be either way. An example commissive would be, “I promise to be there tomorrow for the ceremony”.

Expressives reveal the speaker or sender’s disposition and outlook toward a particular proposition, which he/she communicates through the message. These could be various forms of expression or emotion like congratulatory, sarcastic, and so on. An example expressive would be, “Congratulations on graduating top of the class!”.

Declarations are powerful speech acts since they have the capability to change the reality based on the declared proposition of the message communicated by the speaker/sender. The usual direction of fit is world-to-word, but it can go the other way also. An example declaration would be, "I hereby declare him to be guilty of all charges".

These speech acts are the primary ways in which language is used and communicated among human beings. Without even realizing it, you end up using hundreds of these on any given day. We now look at linguistics and some of the main areas of research associated with it.

Linguistics

We have seen what natural language means, how language is learned and used, and the origins of language acquisition. In fact, a lot of these things are actually formally researched and studied in linguistics by researchers and scholars called *linguists*. Formally, linguistics is defined as the scientific study of language, including the form and syntax of language, the meaning and semantics depicted by the usage of language, and the context of use. The origins of linguistics can be dated back to the 4th century BCE, when Indian scholar and linguist Panini formalized the Sanskrit language description. The term *linguistics* was first defined to indicate the scientific study of languages in 1847 approximately before which the term *philology* was used to indicate the same. While a detailed exploration of linguistics is not needed for text analytics, it is useful to know the different areas of linguistics because some of them are used extensively in natural language processing and text analytics algorithms. The main distinctive areas of study under linguistics are mentioned next.

- **Phonetics:** This is the study of the acoustic properties of sounds produced by the human vocal tract during a speech. This includes studying the sound properties of how they are created as well as perceived by human beings. The smallest individual unit of human speech is termed a *phoneme*, which is usually distinctive to a specific language as opposed to a more generic term, called a *phone*.

- **Phonology:** This is the study of sound patterns as interpreted in the human mind and used for distinguishing between different phonemes. The structure, combination, and interpretations of phonemes are studied in detail, usually by taking into account a specific language at a time. The English language consists of around 45 phonemes. Phonology usually extends beyond just studying phonemes and includes things like accents, tone, and syllable structures.
- **Syntax:** This is usually the study of sentences, phrases, words, and their structures. This includes researching how words are combined grammatically to form phrases and sentences. Syntactic order of words used in a phrase or a sentence matter since the order can change the meaning entirely.
- **Semantics:** This involves the study of meaning in language and can be further subdivided into lexical and compositional semantics.
    - **Lexical semantics:** This involves the study of the meanings of words and symbols using morphology and syntax.
    - **Compositional semantics:** This involves studying relationships among words and combination of words and understanding the meaning of phrases and sentences and how they are related.
- **Morphology:** By definition, a *morpheme* is the smallest unit of language that has distinctive meaning. This includes things like words, prefixes, suffixes, and so on, which have their own distinct meaning. Morphology is the study of the structure and meaning of these distinctive units or morphemes in a language. There are specific rules and syntaxes that govern the way morphemes can combine.
- **Lexicon:** This is the study of properties of words and phrases used in a language and how they build the vocabulary of the language. These include what kinds of sounds are associated with meanings for words, as well as the parts of speech that words belong to and their morphological forms.

- **Pragmatics:** This is the study of how linguistic and non-linguistic factors like context and scenario might affect the meaning of an expression of a message or an utterance. This includes trying to infer if there are any hidden or indirect meanings in communication.
- **Discourse analysis**: This analyzes language and exchange of information in the form of sentences across conversations among human beings. These conversations could be spoken, written, or even signed.
- **Stylistics:** This is the study of language with a focus on the style of writing including the tone, accent, dialogue, grammar, and type of voice.
- **Semiotics:** This is the study of signs, symbols, and sign processes and how they communicate meaning. Things like analogies, metaphors, and symbolism are covered in this area.

While these are the main areas of study and research, linguistics is an enormous field and has a much bigger scope than what is mentioned here. However, things like language syntax and semantics are some of the most important concepts and often form the foundations of natural language processing (NLP). Hence, we look at them in more detail in the following section. We showcase some of the concepts with hands-on examples for better understanding. You can also check out the Jupyter notebook for Chapter 1 in my GitHub repository at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition if you want to follow along and run the examples yourself. Load the following dependencies in your own Python environment to get started. Detailed instructions to install and set up Python and specific frameworks are covered in Chapter 2.

```
import nltk
import spacy
import numpy as np
import pandas as pd

# following line is optional for custom vocabulary installation
# you can use nlp = spacy.load('en')
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)
```

Language Syntax and Structure

We already know what language syntax and structure indicate. Syntax and structure usually go hand in hand where a set of specific rules, conventions, and principles govern the way words are combined into phrases. Phrases are combined into clauses and clauses are combined into sentences. We talk specifically about the English language syntax and structure in this section, since in this book we deal with textual data that belongs to the English language. However, a lot of these concepts can be extended to other languages too. Knowledge about the structure and syntax of language is helpful in many areas, like text processing, annotation, and parsing for further operations like text classification or summarization.

In English, words combine to form other constituent units. These constituents include words, phrases, clauses, and sentences. All these constituents exist together in any message and are related to each other in a hierarchical structure. Moreover, a sentence is a structured format representing a collection of words provided they follow certain syntactic rules like grammar. Let's take a sample sentence, "The brown fox is quick and he is jumping over the lazy dog". The following snippet shows us how the sentence looks in Python.

```
sentence = "The brown fox is quick and he is jumping over the lazy dog"
sentence
```

```
'The brown fox is quick and he is jumping over the lazy dog'
```

The grammar and ordering of words definitely gives meaning to a sentence. What if we jumbled up the words? Would the sentence still make sense?

```
words = sentence.split()
np.random.shuffle(words)
print(words)
```

```
['quick', 'is', 'fox', 'brown', 'The', 'and', 'the', 'is', 'he', 'dog',
'lazy', 'jumping', 'over']
```

This unordered bunch of words, as represented in Figure 1-3, is definitely hard to make sense of, isn't it?

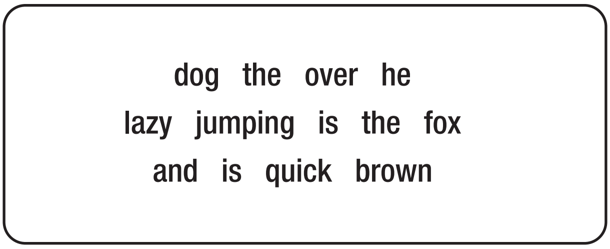


Figure 1-3. *A collection of words without any relation or structure*

From the collection of words in Figure 1-3, it is very difficult to ascertain what it might be trying to convey. Indeed, languages are not just comprised of a bag or bunch of unstructured words. Sentences with proper syntax not only help us give proper structure and relate words, but also help the words convey meaning based on order or position. Considering our previous hierarchy of *sentence* → *clause* → *phrase* → *word*, we can construct the hierarchical sentence tree shown in Figure 1-4 using *shallow parsing*, a technique often used for determining the constituents in a sentence.

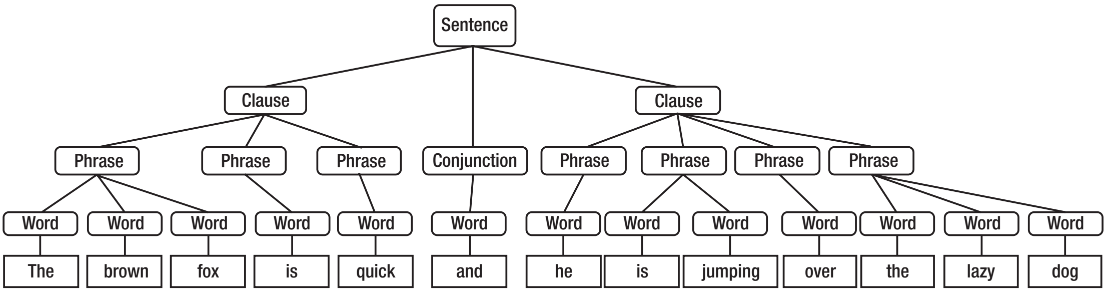


Figure 1-4. *Structured sentence following the hierarchical syntax*

From the hierarchical tree in Figure 1-4, we get the following sentence, "The brown fox is quick and he is jumping over the lazy dog" with a sentence of structure and meaning. We can see that the leaf nodes of the tree consist of words, which are the smallest unit here, and combinations of words form phrases, which in turn form clauses. Clauses are connected through various filler terms or words like conjunctions and they form the final sentence. In the following section, we look at each of these constituents in further detail and learn how to analyze them and determine the major syntactic categories.

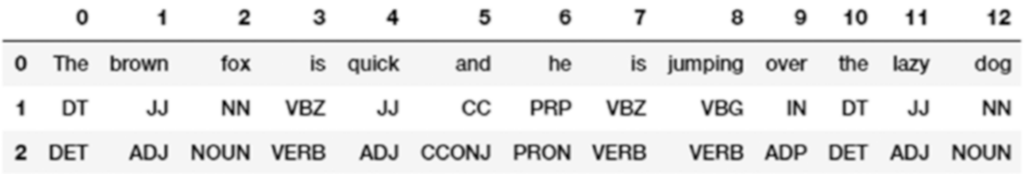
Words

Words are the smallest unit in a language; they are independent and have a meaning of their own. Although morphemes are the smallest distinctive units, they are not independent like words. A word can be comprised of several morphemes. It is useful to annotate and tag words and then analyze them into their parts of speech (POS) to see the major syntactic categories. Here, we cover the main categories and significance of the various POS tags; however, we examine them in further detail and look at methods to generate POS tags programmatically in Chapter 3. Words typically fall into one of the following major categories:

- **N(oun):** This usually denotes words that depict some object or entity that could be living or non-living. Some examples are fox, dog, book, and so on. The POS tag symbol for nouns is `N`.
- **V(erb):** Verbs are words that are used to describe certain actions, states, or occurrences. There are a wide variety of further sub-categories like auxiliary, reflexive, transitive, and many more. Some typical examples of verbs are running, jumping, read, and write. The POS tag symbol for verbs is `V`.
- **Adj(ective):** Adjectives are words that describe or qualify other words, typically nouns and noun phrases. The phrase “beautiful flower” has the noun (`N`) “flower,” which is described or qualified using the adjective (`ADJ`) “beautiful”. The POS tag symbol for adjectives is `ADJ`.
- **Adv(erb):** Adverbs usually act as modifiers for other words including nouns, adjectives, verbs, or other adverbs. The phrase “very beautiful flower” has the adverb (`ADV`) “very,” which modifies the adjective (`ADJ`) “beautiful” indicating the degree of how beautiful the flower is. The POS tag symbol for adverbs is `ADV`.

Besides these four major categories of parts of speech, there are other categories that occur frequently in the English language. These include pronouns, prepositions, interjections, conjunctions, determiners, and many others. Each POS tag, like nouns (`N`) can be further sub-divided into various categories, like singular nouns (`NN`), singular proper nouns (`NNP`), and plural nouns (`NNS`). We look at POS tags in further detail in Chapter 3 when we process and parse textual data and implement POS taggers to annotate text. Considering our previous example sentence, "The brown fox is quick and he is jumping over the lazy dog," we can leverage NLTK or spaCy in Python to annotate it with POS tags. See Figure 1-5.

```
pos_tags = nltk.pos_tag(sentence.split())
pd.DataFrame(pos_tags).T
```

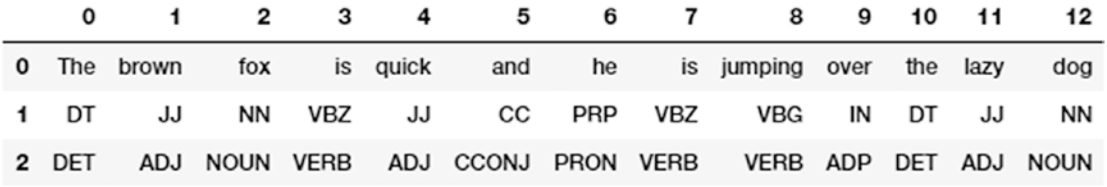


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	The	brown	fox	is	quick	and	he	is	jumping	over	the	lazy	dog
1	DT	JJ	NN	VBZ	JJ	CC	PRP	VBZ	VBG	IN	DT	JJ	NN
2	DET	ADJ	NOUN	VERB	ADJ	CCONJ	PRON	VERB	VERB	ADP	DET	ADJ	NOUN

Figure 1-5. *Annotated words with their parts of speech tags using NLTK*

We talk about the meaning of each POS tag in detail in Chapter 3. But you can still leverage spaCy to understand the high-level semantics of each tag annotation. See Figure 1-6.

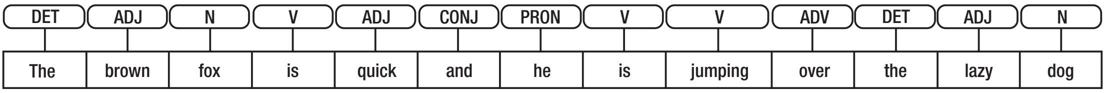
```
spacy_pos_tagged = [(word, word.tag_, word.pos_) for word in nlp(sentence)]
pd.DataFrame(spacy_pos_tagged).T
```



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	The	brown	fox	is	quick	and	he	is	jumping	over	the	lazy	dog
1	DT	JJ	NN	VBZ	JJ	CC	PRP	VBZ	VBG	IN	DT	JJ	NN
2	DET	ADJ	NOUN	VERB	ADJ	CCONJ	PRON	VERB	VERB	ADP	DET	ADJ	NOUN

Figure 1-6. *Annotated words with their parts of speech tags using spaCy*

It is interesting to see that, based on the output depicted in Figure 1-6, the tag annotations match in both frameworks. Internally, they use the Penn Treebank notation for POS tag annotation. Tying this back to our discussion, a simple annotation of our sentence using basic POS tags would look as depicted in Figure 1-7.



DET	ADJ	N	V	ADJ	CONJ	PRON	V	V	ADV	DET	ADJ	N
The	brown	fox	is	quick	and	he	is	jumping	over	the	lazy	dog

Figure 1-7. *Annotated words with their parts of speech tags*

From this example, you might see a few unknown tags. The tag `DET` stands for determiner, which is used to depict articles like a, an, the, etc. The tag `CONJ` indicates a conjunction, which usually bind together clauses to form sentences, and the `PRON` tag stands for pronoun, which are words that represent or take the place of a noun. The `N`, `V`, `ADJ`, and `ADV` tags are typical open classes and represent words belonging to an open vocabulary. Open classes are word classes that consist of an infinite set of words and commonly accept the addition of new words to the vocabulary. Words are usually added to open classes through processes like morphological derivation, invention based on usage, and creating compound lexemes. Some popular nouns that have been added include "internet" and "multimedia". Closed classes consist of a closed and finite set of words and they do not accept new additions. Pronouns are a closed class. In the following section, we look at the next level of the hierarchy, phrases.

Phrases

Words have their own lexical properties like parts of speech, which we saw earlier. Using these words, we can order them in ways that give meaning to the words such that each word belongs to a corresponding phrasal category and one of the words is the main or head word. In the hierarchy tree, groups of words make up phrases, which form the third level in the syntax tree. By principle, phrases are assumed to have at least two or more words considering the pecking order of *words ⟵ phrases ⟵ clauses ⟵ symbols*. However, a phrase can be a single word or a combination of words based on the syntax and position of the phrase in a clause or sentence. For example, the sentence, "Dessert was good" has only three words and each of them roll up to three phrases. The word "dessert" is a noun as well as a noun phrase, "is" depicts a verb as well as a verb phrase, and "good" represents an adjective as well as an adjective phrase describing the aforementioned dessert. There are five major categories of phrases, described next:

- **Noun phrase (NP):** These are phrases where a noun acts as the head word. Noun phrases act as a subject or object to a verb. Noun phrases can be a set of words that can be replaced by a pronoun without rendering the sentence or clause syntactically incorrect. Some examples are "dessert", "the lazy dog", and "the brown fox".

- **Verb phrase (VP):** These phrases are lexical units that have a verb acting as the head word. Usually there are two forms of verb phrases, one form has the verb components as well as other entities like nouns adjectives or adverbs that are a part of the object. The verb here is known as a finite verb. It acts as a single unit in the hierarchy tree and can function as the root in a clause. This form is prominent in constituency grammars. The other form is where the finite verb acts as the root of the entire clause and is prominent in dependency grammars. Another derivation of this includes verb phrases strictly consisting of verb components, including main, auxiliary, infinitive, and participles. The following sentence, "He has started the engine" can be used to illustrate the two types of verb phrases. They would be "has started the engine" and "has started" based on the two forms discussed.
- **Adjective phrase (ADJP):** These are phrases whose head word is an adjective. Their main role is to describe or qualify nouns and pronouns in a sentence and they will be placed before or after the noun or pronoun. The sentence, "The cat is too quick" has an adjective phrase, "too quick" qualifying the cat, which is a noun phrase.
- **Adverb phrase (ADVP):** These phrases act like an adverb since the adverb acts as the head word in the phrase. Adverb phrases are used as modifiers for nouns, verbs, or adverbs by providing further details to describe or qualify them. Considering the sentence, "The train should be at the station pretty soon", the adjective phrase is "pretty soon" since it describes when the train will be arriving.
- **Prepositional phrase (PP):** These phrases usually contain a preposition as the head word and other lexical components like nouns, pronouns, etc. They act like an adjective or adverb describing other words or phrases. The sentence "Going up the stairs" has a prepositional phrase, "up" describing the direction of the stairs.

These five major syntactic categories of phrases can be generated from words using several rules, some of which we discussed, like utilizing syntax and grammar of different types. We explore some of the popular grammars in a later section. Shallow parsing is a popular natural language processing technique used to extract these constituents, including POS tags as well as phrases from a sentence. For our sentence, "The brown fox is quick and he is jumping over the lazy dog," one way of representing it using shallow parsing is to have seven phrases, as depicted in Figure 1-8.

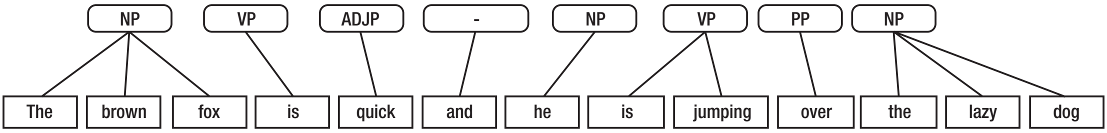


***Figure 1-8.** Annotated phrases with their tags in shallow parsing*

The phrase tags fall into the categories we discussed earlier; however the word "and" is a conjunction and is used to combine clauses. Is there a better way to do this? Probably! You can define your own rules for phrases and then enable shallow parsing or chunking using a lookup based parser similar to NLTK's RegexpParser. It's a grammar based chunk parser and uses a set of regular expression patterns (defined grammar rules) to specify the behavior of the parser. The following code shows it in action for our sentence! See Figure 1-9 too.

```
grammar = '''
            NP: {<DT>?<JJ>?<NN.*>}
            ADJP: {<JJ>}
            ADVP: {<RB.*>}
            PP: {<IN>}
            VP: {<MD>?<VB.*>+}
          '''

pos_tagged_sent = nltk.pos_tag(sentence.split())
rp = nltk.RegexpParser(grammar)
shallow_parsed_sent = rp.parse(pos_tagged_sent)
print(shallow_parsed_sent)
```

```
(S
  (NP The/DT brown/JJ fox/NN)
  (VP is/VBZ)
  (ADJP quick/JJ)
  and/CC
  he/PRP
  (VP is/VBZ jumping/VBG)
  (PP over/IN)
  (NP the/DT lazy/JJ dog/NN))

# visualize shallow parse tree
shallow_parsed_sent
```

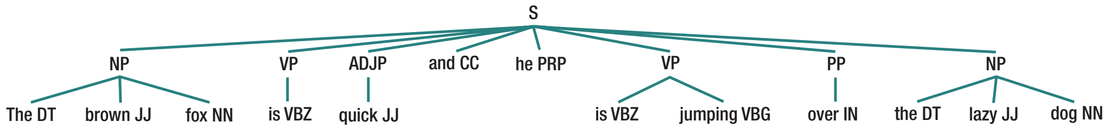


Figure 1-9. *Annotated phrases with their tags in shallow parsing using NLTK*

Based on the output in Figure 1-9, we can see the power of a simple rule-based chunker in identifying major phrases and sentence structure. In the following section, we look at clauses, their main categories, and some conventions and syntactic rules for extracting clauses from sentences.

Clauses

By nature, clauses can act as independent sentences or several clauses can be combined together to form a sentence. Clauses are groups of words with some relation between them and they usually contain a subject and a predicate. Sometimes the subject may not be present and the predicate has a verb phrase or a verb with an object. By default you can classify clauses into two distinct categories, the main clause and the subordinate clause. The main clause is also known as an independent clause because it can form a sentence by itself and act as both a sentence and a clause. The subordinate or dependent clause cannot exist by itself; it depends on the main clause for its meaning. They are usually with other clauses using dependent words like subordinating conjunctions.

Since we are talking a lot about syntactic properties of language, clauses can be subdivided into several categories based on syntax. They are explained in detail as follows.

- **Declarative:** These clauses occur quite frequently and denote statements that do not have a specific tone associated with them. These are just standard statements that are declared with a neutral tone and could be factual or non-factual. An example would be, “Grass is green”.

- **Imperative:** These clauses are usually in the form of a request, command, rule, or advice. The tone in this case is a person issuing an order to one or more people to carry out an order, request, or even an instruction. An example would be, “Please do not talk in class”.
- **Relative:** The simplest interpretation of relative clauses is that they are subordinate clauses and hence dependent on another part of the sentence, which usually contains a word, phrase, or even a clause. This element usually acts as the antecedent to one of the words from the relative clause and relates to it. A simple example would be the following sentence, “John just mentioned that he wanted a soda”. The antecedent is the proper noun, “John” which was referred to in the relative clause, “he wanted a soda”.
- **Interrogative:** These clauses are typically in the form of questions. The type of these questions can be either affirmative or negative. Some example would be, “Did you get my mail?” and “Didn’t you go to school?”
- **Exclamative:** These clauses are used to express shock, surprise, or even compliments. All these expressions fall under exclamations and these clauses often end with an exclamation mark. An example is “What an amazing race!”

Most clauses are expressed in one of these syntactic forms; however, this list of clause categories is not exhaustive and can be further categorized into several other forms. Considering our example sentence, “The brown fox is quick and he is jumping over the lazy dog,” if you remember the syntax tree, the coordinating conjunction (and) divides the sentence into two clauses. They are “The brown fox is quick” and “he is jumping over the lazy dog”. Can you guess what categories they might fall into? (Hint: Look back at the definitions of declarative and relative clauses.)

Grammar

Grammar helps enable syntax and structure in language. It primarily consists of a set of rules that are used to determine how to position words, phrases, and clauses when constructing sentences in a natural language. Grammar is not restricted to the written word but is also used verbally. These rules can be specific to a region, language, or dialect, or be somewhat universal like the Subject-Verb-Object (SVO) model. Origins of grammar have a rich history, starting with Sanskrit in India. In the West, the study of grammar originated with the Greeks and the earliest work was the *Art of Grammar*, written by Dionysius Thrax. Latin grammar models were developed from the Greek models and gradually across several ages, grammar models for various languages started being created. It was only in the 18th Century that grammar was considered a serious candidate for being a field under linguistics.

Grammar was developed over the course of time and has kept evolving leading to the birth of newer types of grammar. Hence, grammar is not a fixed set of rules but evolves based on language use over the course of time. Considering the English language as before, there are several ways that grammar can be classified. We first talk about two broad classes into which most of the popular grammatical frameworks can be grouped and then we further explore how these grammar frameworks represent language. Grammar can be subdivided into two main classes based on its representations for linguistic syntax and structure. They are as follows:

- Dependency grammar
- Constituency grammar

Dependency Grammar

This is a class of grammar that specifically does not focus on constituents (unlike constituency grammars) like words, phrases, and clauses, but gives more emphasis on words. Hence these grammar types are also known as word-based grammars. To understand dependency grammar, we should first know what dependency means in this context. Dependencies in this context are labeled word-word relations or links that are usually asymmetrical. A word has a relation or depends on another word based on the positioning of the words in the sentence. Consequently, dependency grammars assume that further constituents of phrases and clauses are derived from this dependency structure between words. The basic principle behind dependency grammar is that in any sentence in the language, all the words except one has some relationship or dependency on other words in the sentence. The word that has no dependency is termed the *root* of the sentence. The verb is taken as the root of the sentence in most cases. All the other words are directly or indirectly linked to the root verb using links, which are the dependencies. While there are no concepts of phrases or clauses, looking at the syntax and relations between words and their dependents, one can determine the necessary constituents in the sentence.

Dependency grammars always have a one-to-one relationship to each word in the sentence. There are two aspects to this grammar representation. One is the syntax or structure of the sentence and the other is the semantics obtained from the relationships denoted between the words. The syntax or structure of the words and their interconnections can be shown using a sentence syntax or parse tree, similar to what we depicted in an earlier section. Considering our sentence, “The brown fox is quick and he is jumping over the lazy dog,” if we would want to draw the dependency syntax tree for this, we would have the structure shown in Figure 1-10.

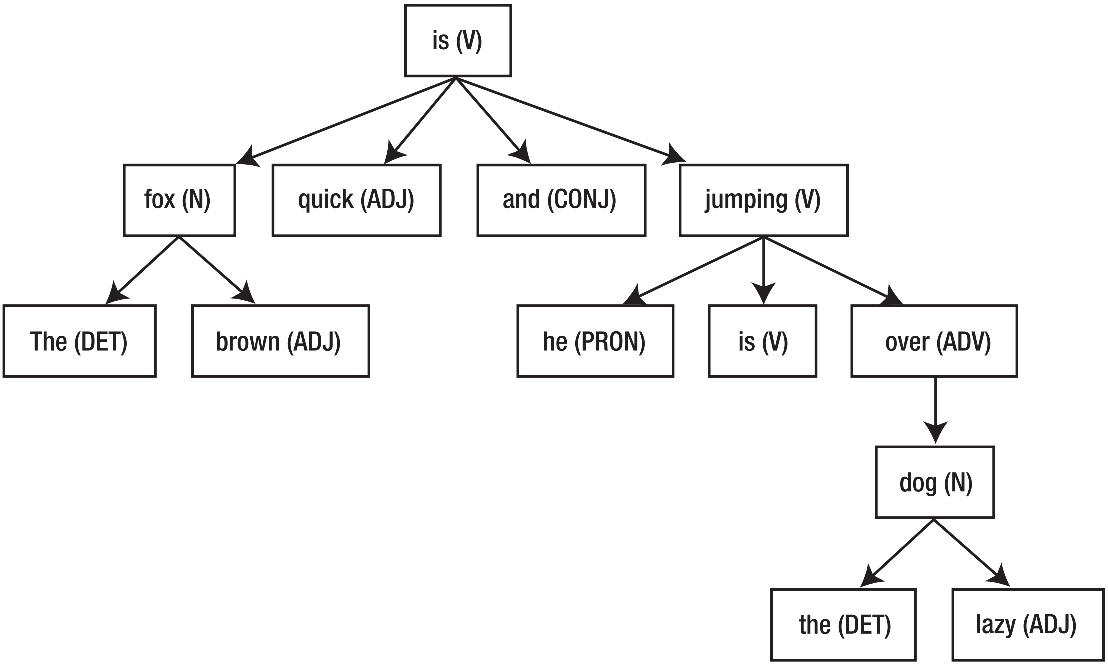


Figure 1-10. *Dependency grammar based syntax tree with POS tags*

Figure 1-10 shows us that the dependencies form a tree or, to be more accurate, a graph over all the words in the sentence. The graph is connected where each word has at least one directed edge going out or coming into it. The graph is also directed since each edge between two words points in one specific direction. Hence, the dependency tree is a directed acyclic graph (DAG). Every node in the tree has at most one incoming edge except the root node. Since this is a directed graph, by nature, dependency trees do not depict the order of the words in the sentence. The emphasis is more on the relationship between the words in the sentence. Our sentence is annotated in Figure 1-10 with the relevant POS tags, which we discussed earlier, and the directed edges showing the dependency.

Now, if you remember, we discussed earlier that there were two aspects to the representation of sentences using dependency grammar. Each directed edge represents a specific type of meaningful relationship (also known as a syntactic function) and we can annotate our sentence, further showing the specific dependency relationship types between the words. This is depicted in Figure 1-11.

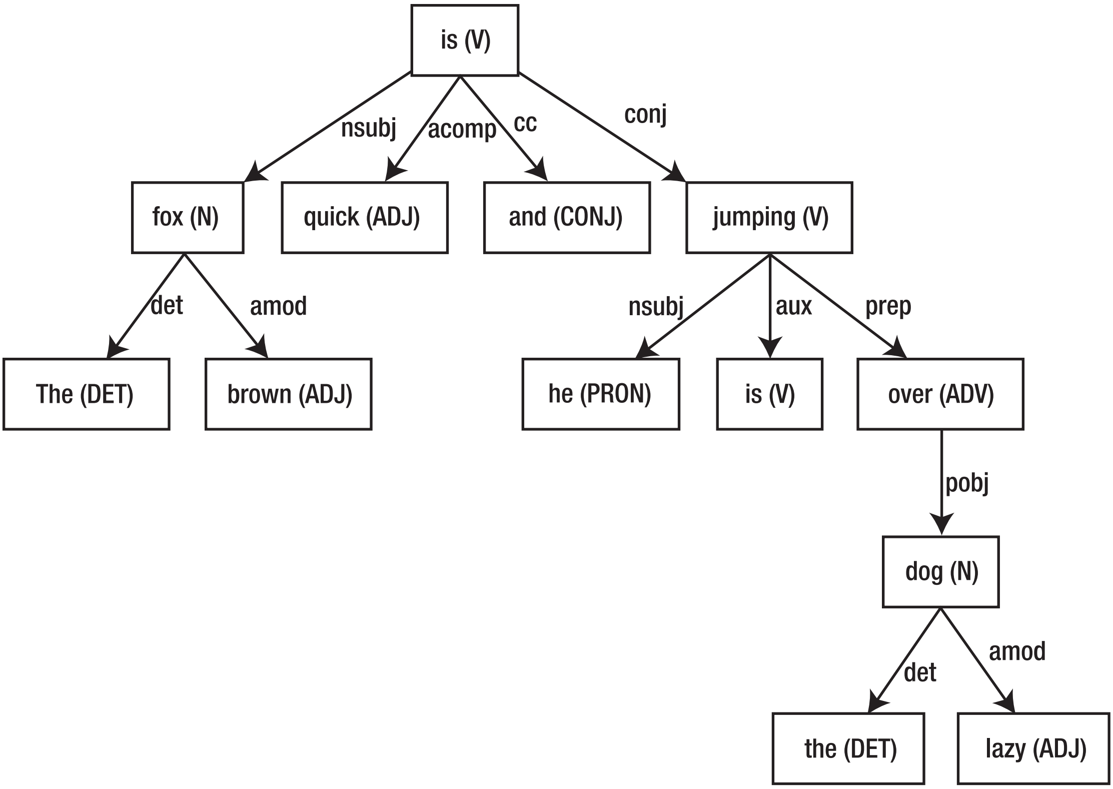


***Figure 1-11.** Dependency grammar based syntax tree annotated with dependency relationship types*

These dependency relationships each have their own meaning and are a part of a list of universal dependency types. This is a part of the original paper, "Universal Stanford Dependencies: A Cross-Linguistic Typology" (de Marneffe et al., 2014). You can check out the exhaustive list of dependency types and their meanings at `http://universaldependencies.org/u/dep/index.html`. The spaCy framework leverages this universal dependency scheme and an alternate scheme called the CLEAR dependency scheme. Details on possible dependencies are available at `https://emorynlp.github.io/nlp4j/components/dependency-parsing.html` in case you are interested in understanding the significance of all these dependencies. If we observe some of these dependencies, it is not too hard to understand them. We look in detail some of the tags used in the dependencies for the sentence in Figure 1-11.

- The dependency tag `det` denotes the determiner relationship between a nominal head and the determiner. Usually the word with POS tag `DET` will also have the `det` relation. Examples include (`fox -> the`) and (`dog -> the`).
- The dependency tag `amod` stands for adjectival modifier and stands for any adjective that modifies the meaning of a noun. Examples include (`fox -> brown`) and (`dog -> lazy`).
- The dependency tag `nsubj` stands for an entity that acts as a subject or agent in a clause. Examples include (`is -> fox`) and (`jumping -> he`).
- The dependencies `cc` and `conj` are linkages related to words connected by coordinating conjunctions. Examples include (`is -> and`) and (`is -> jumping`).
- The dependency tag `aux` indicates the auxiliary or secondary verb in the clause. Examples include (`jumping -> is`).
- The dependency tag `acomp` stands for adjective complement and acts as the complement or object to a verb in the sentence. Examples include (`is -> quick`).
- The dependency tag `prep` denotes a prepositional modifier that usually modifies the meaning of a noun, verb, adjective, or even a preposition. This representation is used for prepositions having a noun or noun phrase complement. Examples include (`jumping -> over`).
- The dependency tag `pobj` is used to denote the object of a preposition. This is usually the head of a noun phrase following a preposition in the sentence. Examples include (`over -> dog`).

These tags have been extensively used in our sample sentence for annotating the various dependency relationships among the words. Now that you understand dependency relationships better, it would be good to remember that often when representing a dependency grammar for sentences, instead of creating a tree with linear orders, you can also represent it with a normal graph since there is no concept of order of words in dependency grammar. We can leverage spaCy to build this dependency tree/graph for our sample sentence (see Figure 1-12).

```
from spacy import displacy

displacy.render(nlp(sentence), jupyter=True,
                options={'distance': 100,
                         'arrow_stroke': 1.5,
                         'arrow_width': 8})
```

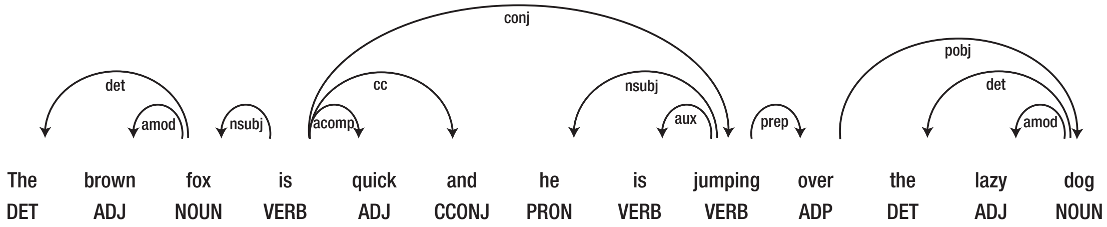


***Figure 1-12.** Dependency grammar annotated graph for our sample sentence with spaCy*

Figure 1-12 depicts our sentence annotated with dependency tags, which should be clear to you based on our earlier discussion. When we cover constituency based grammar next, you will observe that the number of nodes in dependency grammars will be a lot fewer corresponding to their constituency counterparts. Currently there are various grammatical frameworks based on dependency grammar. Some popular ones are algebraic syntax and operator grammar. Next, we look at the concepts behind constituency grammars and their representations.

Constituency Grammar

These grammar types are built on the principle that a sentence can be represented by several constituents derived from it. These grammar types can be used to model or represent the internal structure of sentences in terms of a hierarchically ordered structure of its constituents. Each word belongs to a specific lexical category and forms the head words of different phrases. These phrases are formed based on rules called phrase structure rules. Hence constituency grammars are also called *phrase structure grammars*. Phrase structure grammars were introduced by Noam Chomsky in the 1950s. To understand constituency grammars, we must know clearly what we mean by constituents. We covered this several times earlier in this chapter, but just to refresh your memory, constituents are words or group of words that have specific meaning and can act together as a dependent or independent unit. They also can be combined to form higher order structures in a sentence. These include phrases and clauses.

Phrase structure rules form the core of constituency grammars since they talk about syntax and rules, which govern the hierarchy and ordering of the various constituents in the sentences. These rules cater to two things primarily. First and foremost, these rules determine what words are used to construct the phrases or constituents and secondly, these rules determine how we need to order these constituents. If we want to analyze phrase structure, we should be aware of typical schema patterns of the phrase structure rules. The generic representation of a phrase structure rule is $S \rightarrow A\ B$, which depicts that the structure S consists of constituents A and B and the ordering is A followed by B.

There are several phrase structure rules and we explore them one by one to understand how exactly we extract and order constituents in a sentence. The most important rule describes how to divide a sentence or a clause. The phrase structure rule denotes a binary division for a sentence or a clause as $S \rightarrow NP\ VP$, where S is the sentence or clause and it is divided into the subject, denoted by the noun phrase (NP) and the predicate, denoted by the verb phrase (VP). Of course, we can apply additional rules to break down each of the constituents further, but the top level of the hierarchy starts with a NP and VP. The rule for representing a noun phrase is of the form $NP \rightarrow [DET][ADJ]N\ [PP]$, where the square brackets […] denote that it is optional. A noun phrase usually consists of a (N)oun as the head word and may optionally contain (DET) erminants and (ADJ)ectives describing the noun and a prepositional phrase (PP) at the right side in the syntax tree. Consequently, a noun phrase may contain another noun phrase as a constituent of it. Figure 1-13 shows a few examples that are governed by the aforementioned rules for noun phrases.

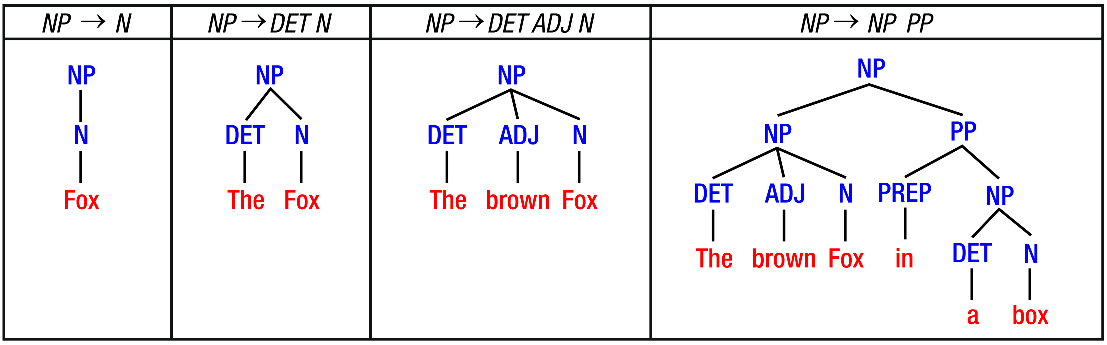


***Figure 1-13.** Constituency syntax trees depicting structuring rules for noun phrases*

These syntax trees shows us the various constituents that a noun phrase typically contains. As mentioned, a noun phrase denoted by NP on the left side of the production rule may also appear on the right side of the production rule, as depicted in the last example. This is a property called *recursion* and we talk about it toward the end of this section. We now look at rules for representing verb phrases. The rule is of the form $VP \rightarrow V \mid MD\ [VP][NP][PP][ADJP][ADVP]$, where the head word is usually a (V)erb or a modal (MD). A modal is itself an auxiliary verb, but we give it a different representation just to distinguish it from the normal verb. This is optionally followed by another verb phrase (VP) or noun phrase (NP), prepositional phrase (PP), adjective phrase (ADJP), or adverbial phrase (ADVP). The verb phrase is always the second component when we split a sentence using the binary division rule, making the noun phrase the first component. Figure 1-14 depicts a few examples for the different types of verb phrases that can be typically constructed and their representations as syntax trees.

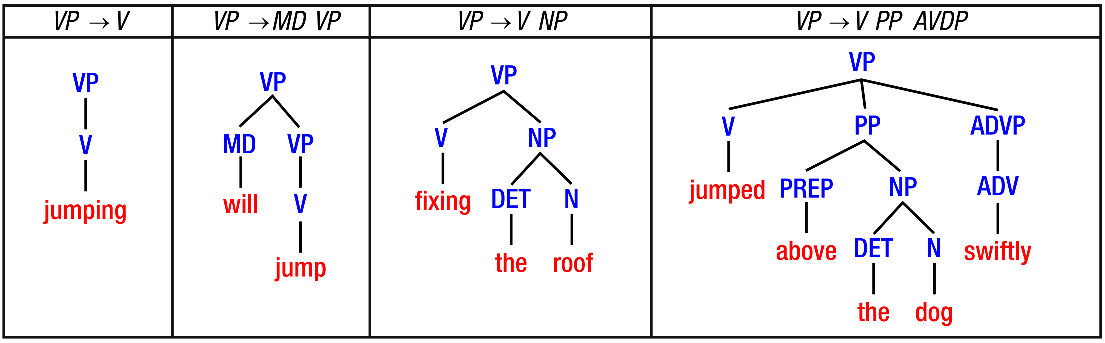


***Figure 1-14.** Constituency syntax trees depicting structuring rules for verb phrases*

Like we depicted earlier, these syntax trees show us the representations of the various constituents in verb phrases and, using the property of recursion, a verb phrase may also contain another verb phrase inside it, as we see in the second syntax tree. We also see the hierarchy being maintained specially in the third and fourth syntax trees, where the NP and PP by itself are further constituents under the VP and they can be further broken down into smaller constituents. Since we have seen a lot of prepositional phrases being used in the previous examples, let's look at the production rules for representing prepositional phrases. The basic rule has the form *PP* → *PREP* [*NP*], where PREP denotes a preposition that acts as the head word and it is optionally followed by a noun phrase (NP). Figure 1-15 depicts some representations of prepositional phrases and their corresponding syntax trees.

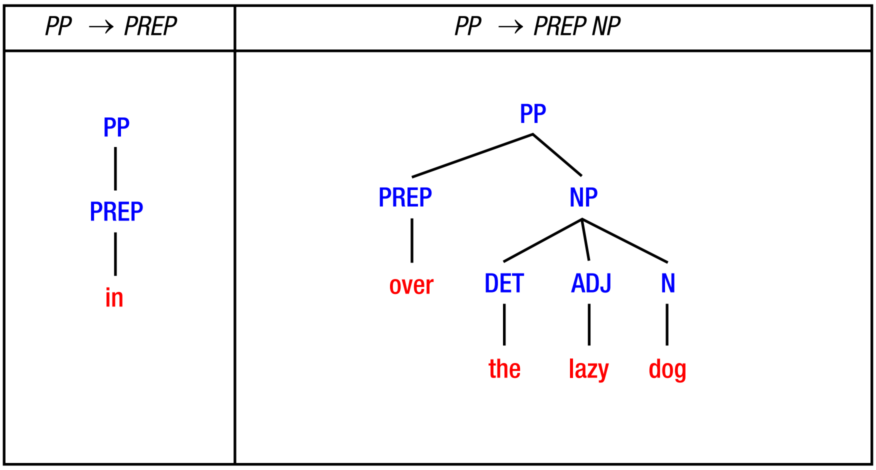


***Figure 1-15.** Constituency syntax trees depicting structuring rules for prepositional phrases*

These syntax trees show us some different representations for prepositional phrases. We now discuss the concept of recursion. Recursion is an inherent property of the language that allows constituents to be embedded in other constituents, which are depicted by different phrasal categories that appear on both sides of the production rules. This enables us to create long constituency-based syntax trees from sentences. A simple example is the representation of the sentence, "The flying monkey in the circus on the trapeze by the river" depicted by the constituency parse tree in Figure 1-16.

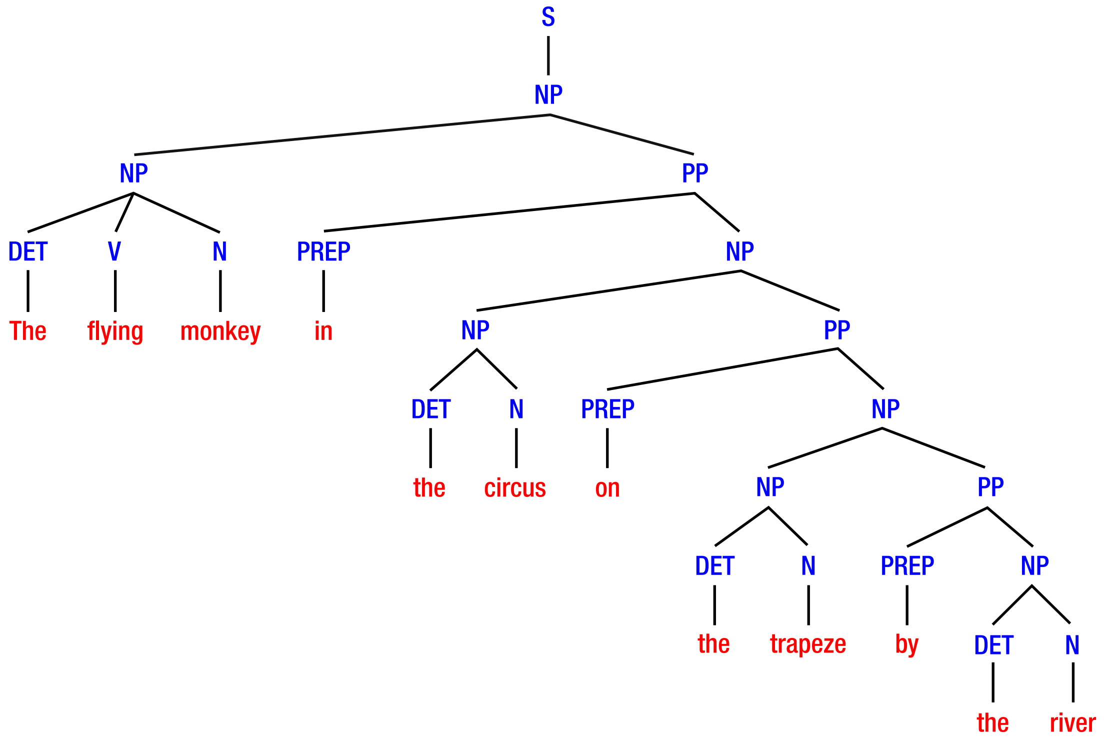


Figure 1-16. *Constituency syntax tree depicting recursive properties among constituents*

If you closely observe the syntax tree in Figure 1-16, you will notice that it is made up of only noun phrases and prepositional phrases. However, due to the inherent recursive property that a prepositional phrase itself can consist of a noun phrase and the noun phrase can consist of a noun phrase as well as a prepositional phrase, the hierarchical structure has multiple NPs and PPs. If you go over the production rules for noun phrases and prepositional phrases, you will find the constituents in the tree in Figure 1-16 adhere to the rules.

We now talk a bit about conjunctions, since they are used to join clauses and phrases and form an important part of language syntax. Usually words, phrases and even clauses can be combined using conjunctions. The production rule can be denoted as $S \rightarrow S\, conj\, S \;\forall\; S \in \{S, NP, VP\}$, where two constituents can be joined by a conjunction denoted by `conj` in the rule. A simple example for a sentence consisting of a noun phrase which by itself is constructed out of two noun phrases and a conjunction would be, "The brown fox and the lazy dog". This is depicted by the constituency syntax tree showing the adherence to the production rule in Figure 1-17.

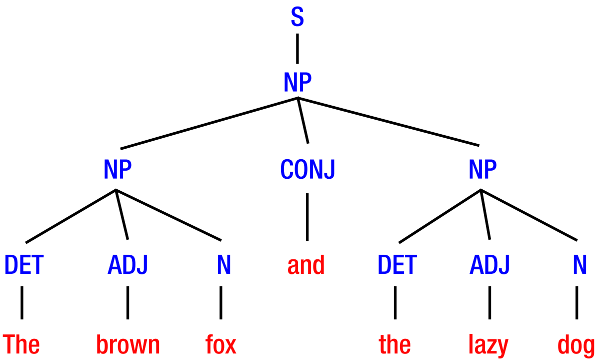


Figure 1-17. *Constituency syntax tree depicting noun phrases joined by a conjunction*

Figure 1-17 shows us that the top-level noun phrase is the sentence by itself and it has two noun phrases as its constituents. They are joined by a conjunction, thus satisfying our aforementioned production rule. What if we wanted to join two sentences or clauses together with a conjunction? We can do that by putting all of these rules and conventions together and generating the constituency based syntax tree for our sample sentence, "The brown fox is quick and he is jumping over the lazy dog". This would give us the syntactic representation of our sentence as depicted in Figure 1-18.

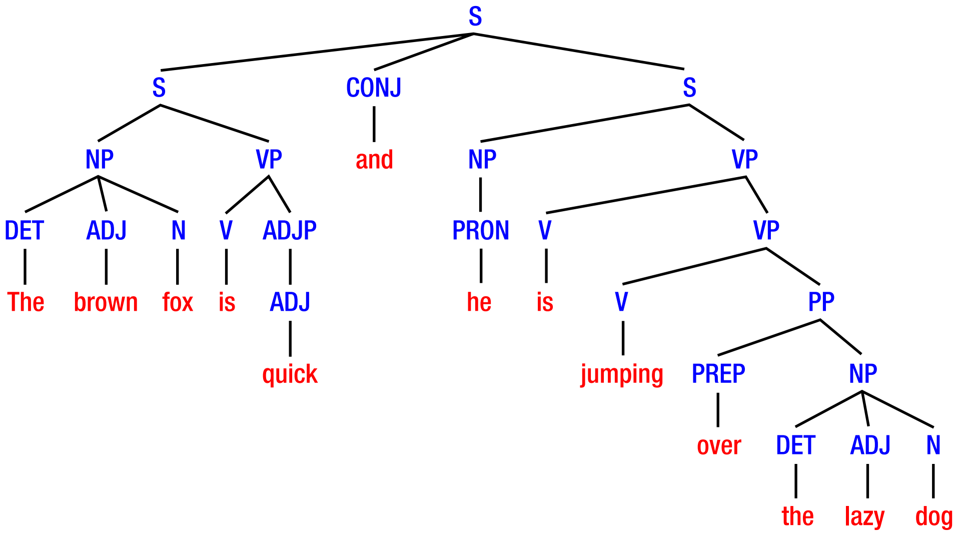


Figure 1-18. *Constituency syntax tree for our sample sentence*

From Figure 1-18, you can conclude that our sentence has two main clauses or constituents, which we had talked about earlier, and they are joined by a coordinating conjunction (and). Moreover, the constituency grammar-based production rules break down the top-level constituents into further constituents consisting of phrases and their words. Looking at this syntax tree, you can see that it does show the order of the words in the sentence and it more of a hierarchical tree-based structure with undirected edges. Hence, this is a lot different compared to the dependency grammar based syntax tree/graph with unordered words and directed edges. There are several popular grammar frameworks based on concepts derived from constituency grammar. These include Phrase Structure Grammar, Arc Pair Grammar, Lexical Functional Grammar, and even the famous Context-Free Grammar, which is used extensively in describing formal language. We can leverage Stanford's Core NLP-based parsers in NLTK to perform constituency parsing on our sample sentence.

```
from nltk.parse.stanford import StanfordParser

scp = StanfordParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar', path_to_models_jar='E:/stanford/
stanford-parser-full-2015-04-20/stanford-parser-3.5.2-models.jar')

result = list(scp.raw_parse(sentence))
print(result[0])
```

```
(ROOT
  (NP
    (S
      (S
        (NP (DT The) (JJ brown) (NN fox))
        (VP (VBZ is) (ADJP (JJ quick))))
      (CC and)
      (S
        (NP (PRP he))
        (VP
          (VBZ is)
          (VP
            (VBG jumping)
            (PP (IN over) (NP (DT the) (JJ lazy) (NN dog))))))))

# visualize constituency tree
result[0]
```

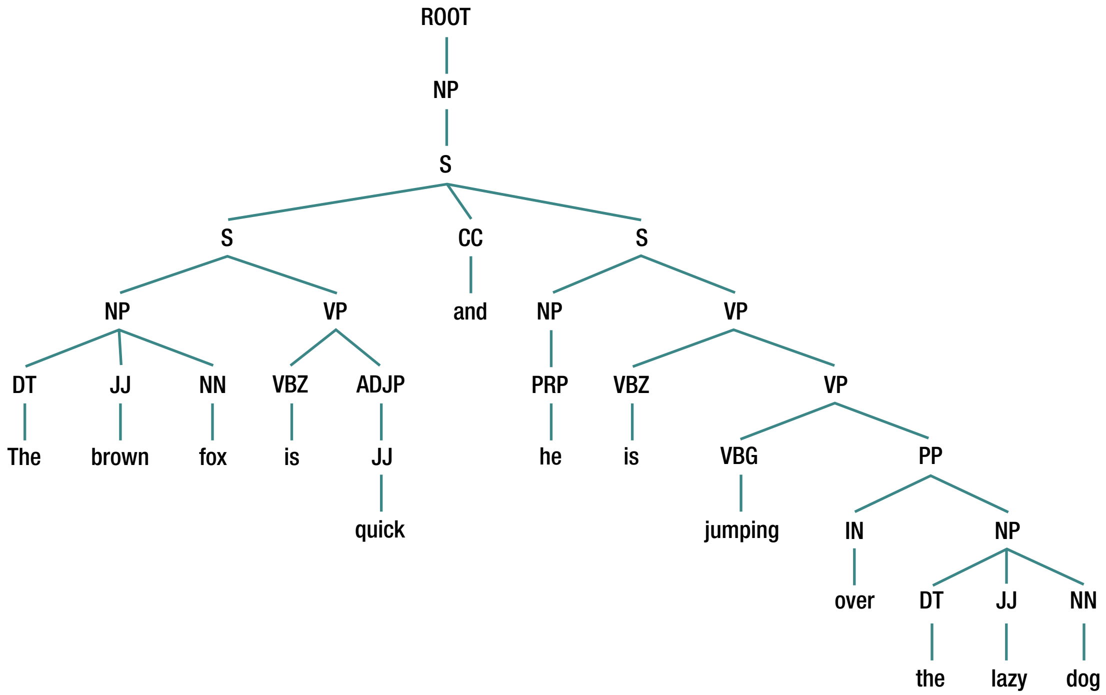


***Figure 1-19.** Constituency syntax tree for our sample sentence with NLTK and Stanford Core NLP*

We can see that the constituency tree depicted in Figure 1-19 has the same hierarchical structure as the tree in Figure 1-18.

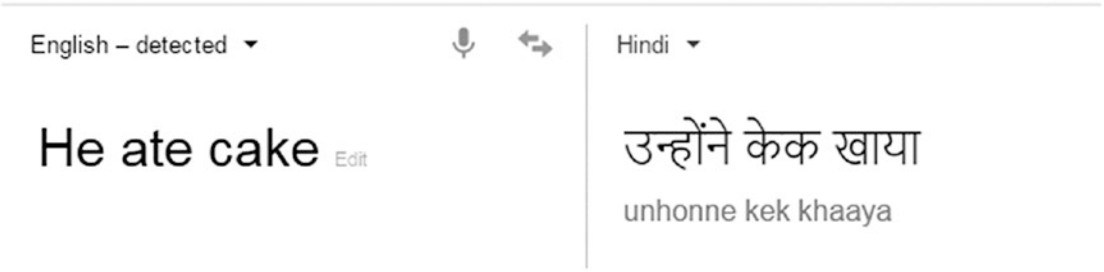
Word-Order Typology

Typology in linguistics is a field that specifically deals with trying to classify languages based on their syntax, structure, and functionality. They can be classified in several ways and one of the most common models is to classify them according to their dominant word orders, also known as *word-order typology*. The primary word orders of interest occur in clauses, consisting of a subject, verb, and an object. Of course, not all clauses will use the subject, verb, and object format and the subject and object are not used in certain languages. However, there exist several different classes of word orders that can be used to classify a wide variety of languages. A survey was done by Russell Tomlin in 1986 and Table 1-1 shows some insights derived from his analysis.

Table 1-1. *Word Order Based Language Classification Surveyed by Russell Tomlin, 1986*

Sl No.	Word Order	Language Frequency	Example Languages
1	Subject-Object-Verb	180 (45%)	Sanskrit, Bengali, Gothic, Hindi, Latin
2	Subject-Verb-Object	168 (42%)	English, French, Mandarin, Spanish
3	Verb-Subject-Object	37 (9%)	Hebrew, Irish, Filipino, Aramaic
4	Verb-Object-Subject	12 (3%)	Baure, Malagasy, Aneityan
5	Object-Verb-Subject	5 (1%)	Apalai, Hixkaryana, Arecua
6	Object-Subject-Verb	1 (0%)	Warao

From this table, we can observe that there are six major classes of word orders and languages like English follow the Subject-Verb-Object word order class. A simple example would be the sentence, “He ate cake” where “He” is the subject, “ate” is the verb, and “cake” is the object. The majority of languages from Table 1-1 follow the Subject-Object-Verb word order. In that case, the sentence, “He cake ate” would be correct if it was translated to those languages. This is illustrated by the English to Hindi translation of the same sentence in Figure 1-20, courtesy of Google Translate.



He ate cake

उन्होंने केक खाया

unhonne kek khaaya

Figure 1-20. *English to Hindi translation changes the word order class for the sentence*

Even if you do not understand Hindi, you can understand by the English annotation provided by Google that the word “cake” denoted by “kek” in the text under the Hindi translation has moved to the middle of the sentence and the verb “ate” denoted by “khaaya” has moved to the end of the sentence, thus making the word order class Subject-Object-Verb, which is the correct form for the Hindi language. This gives us an indication of the importance of word order and how representation of messages can be grammatically different in various languages. This brings us to the end of our discussion regarding the syntax and structure of languages. Next, we look at some of the concepts around language semantics.

Language Semantics

The definition of *semantics* is the study of meaning. Linguistics has its own sub-field of linguistic semantics that deals with the study of meaning in language, including the relationship between words, phrases, and symbols. It studies their indication, meaning, and representation of the knowledge they signify. In simple words, semantics is more concerned with facial expressions, signs, symbols, body language, and knowledge that’s transferred when passing messages from one entity to another. Representing semantics using formal rules or conventions has always been a challenge in linguistics. However, there are different ways to represent meaning and knowledge obtained from language. In the following section, we look at relationships between the lexical units of a language, which are predominantly words and phrases, and explore several concepts around formalizing the representation of knowledge and meaning.

Lexical Semantic Relations

Lexical semantics is concerned with identifying semantic relations between lexical units in a language and how they are correlated to the syntax and structure of the language. Lexical units are usually represented by morphemes, the smallest meaningful and syntactically correct unit of a language. Words are inherently a subset of these morphemes. Each lexical unit has its own syntax, form, and meaning. They also derive meaning from their surrounding lexical units in phrases, clauses, and sentences. A *lexicon* is a complete vocabulary of these lexical units. We explore some concepts revolving around lexical semantics in this section.

Lemmas and Wordforms

A *lemma* is also known as the canonical or citation form for a set of words. The lemma is usually the base form of a set of words, known as a *lexeme*. Lemma is the specific base form or head word that represents the lexeme. Word forms are inflected forms of the lemma, which are part of the lexeme and can appear as one of the words from the lexeme in text. A simple example is the lexeme {eating, ate, eats}, which are the word forms and their lemma is the word "eat".

These words have specific meanings based on their position among other words in a sentence. This is also known as "sense" of the word or wordsense. *Wordsense* gives us a concrete representation of the different aspects of a word's meaning. Consider the word "fair" in the following sentences: "They are going to the annual fair" and "I hope the judgment is fair to all". Even though the word "fair" is the same in both the sentences, the meaning changes based on the surrounding words and context.

Homonyms, Homographs, and Homophones

Homonyms are defined as words that share the same spelling or pronunciation but have different meanings. An alternative definition restricts the constraint to the same spelling. The relationship between these words is termed homonymy. Homonyms are often said to be a superset of homographs and homophones. An example of homonyms can be demonstrated in the following sentences, "The bat hangs upside down from the tree" and "That baseball bat is really sturdy" for the word "bat".

Homographs are defined as words that have the same written form or spelling but have different meanings. Several alternate definitions say that the pronunciation can also be different. Some examples of homographs include, the word "lead" as in "I am using a lead pencil" and "Please lead the soldiers to the camp" and the word "bass" as in "Turn up the bass for the song" and "I just caught a bass today while I was out fishing". Note that with the case of the word "lead," the spelling stays the same but the pronunciation changes based on the context in the sentences.

Homophones are defined as words that have the same pronunciation but have different meanings and can have the same or different spellings. Examples are the words "pair" (which means couple) and "pear" (which means the fruit), which sound the same but have different meanings and written forms. These words cause problems in natural language processing, since it is very difficult to determine the actual context and meaning using machine intelligence.

Heteronyms and Heterographs

Heteronyms are defined as words that have the same written form or spelling but have different pronunciation and meaning. By nature, they are a subset of homographs. They are also often called heterophones, which means, "different sound". Examples of heteronyms are the words "lead" (metal, command) and "tear" (rip off something, moisture from eyes).

Heterographs are defined as words that have the same pronunciation but different meanings as well as spellings. By nature they are a subset of homonyms. Their written representation might be different but they sound very similar or often exactly the same when spoken. Some examples include the words "to," "too," and "two," which sound similar but have different spellings and meanings.

Polysemes

Polysemes are defined as words that have the same written form or spelling and different but related meanings. While this is very similar to homonymy, the difference is very subjective and depends on the context since these words are related to each other. A very good example is the word "bank," which can mean (1) a financial institution, (2) the bank of the river, (3) the building that belongs to the financial institution, or (4) even as a verb which means to rely upon. Now all these examples use the same word, "bank" and are homonyms. But only (1), (3), and even (4), which stand for trust and security (which a financial organization represents) are polysemes and represent a common theme.

Capitonyms

Capitonyms are defined as words that have the same written form or spelling but have different meanings when they are capitalized. The words may or may not have different pronunciations. Some examples include the words "march" ("March" indicates the month and "march" depicts the action of walking) and "may" ("May" indicates the month and "may" depicts a modal verb).

Synonyms and Antonyms

Synonyms are defined as words that have different pronunciations and spellings but have the same meanings in some or all contexts. If two words or lexemes are synonyms, they can be substituted for each other in various contexts and it signifies them having the same propositional meaning. Words that are synonyms are said to be *synonymous* with each other and the state of being a synonym is called *synonymy*. Perfect synonymy is almost non-existent. The reason is because synonymy is more of a relation between senses and has contextual meaning rather than just words. Consider the words "big," "huge," and "large," which are synonyms of each other. They are related and make perfect sense in sentences like, "That milkshake is really (big\large\huge)". However, if we consider the sentence, "Bruce is my big brother," it does not make sense if we substitute the word big with either huge or large. The reason is because the word big here has a context or sense depicting being older and the other two synonyms lack this sense. Synonyms can exist for all parts of speech, including nouns, adjectives, verbs, adverbs, and prepositions.

Antonyms are defined as pairs of words that define a binary opposite relationship. These words indicate specific sense and meaning that are completely opposite of each other. The state of being an antonym is termed *antonymy*. There are three types of antonyms—graded antonyms, complementary antonyms, and relational antonyms. Graded antonyms are antonyms with a certain grade or level when measured on a continuous scale, like the pair (*fat*, *skinny*). Complementary antonyms are word pairs that are opposite in their meaning but they cannot be measured on any grade or scale. An example of a complementary antonym pair is (*divide*, *unite*). Relational antonyms are word pairs that have some relationship between them and the contextual antonymy is signified by this very relationship. An example of a relational antonym pair is (*doctor*, *patient*).

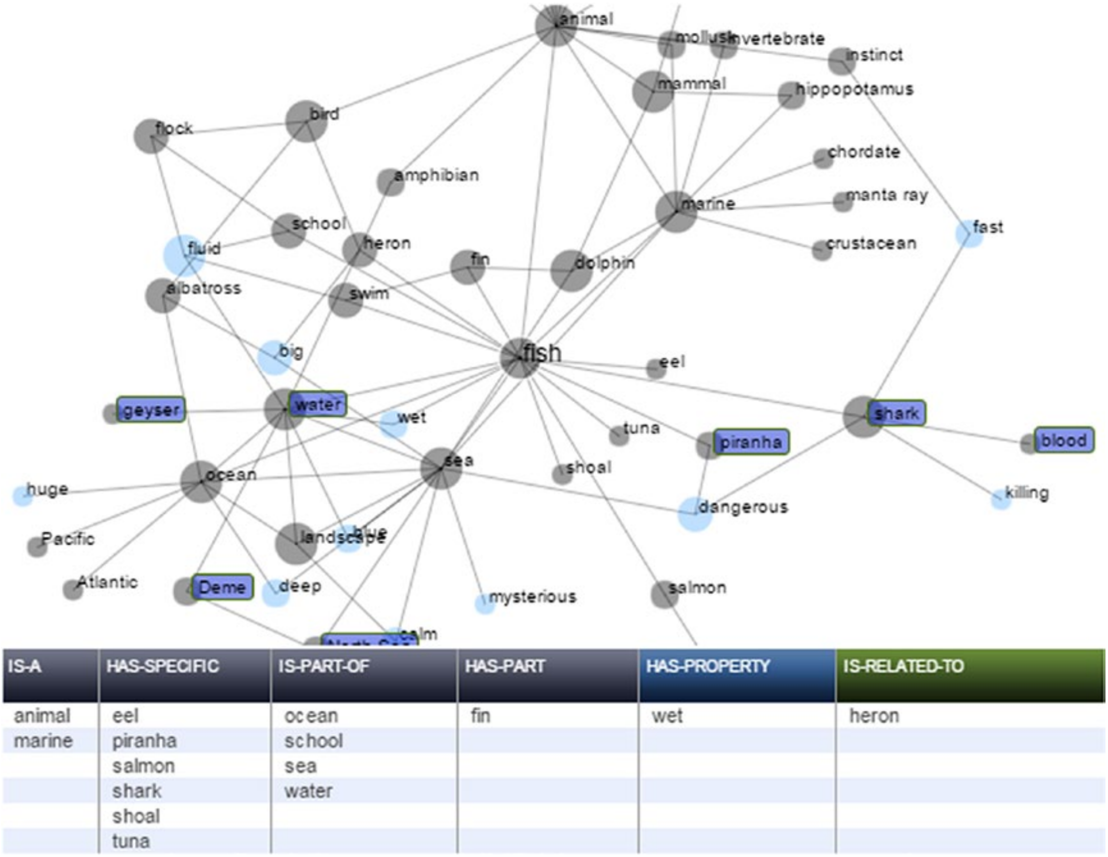
Hyponyms and Hypernyms

Hyponyms are words that are subclasses of other words. In this case the hyponyms are generally words with a very specific sense and context as compared to the word that is their superclass. *Hypernyms* are the words that act as the superclass to the hyponyms and have a more generic sense compared to the hyponyms. An example is the word "fruit," which is the hypernym and the words "mango," "orange," and "pear" are possible hyponyms. The relationship depicted between these words is often called hyponymy or hypernymy.

Semantic Networks and Models

We have seen several ways to formalize relations between words and their sense or meaning. Considering lexical semantics, there are approaches to determine the sense and meaning of each lexical unit, but what if we wanted to consider representing the meaning of some concept or theory that involves relating these lexical units together and forming connections between them based on their meaning? Semantic networks aim to tackle this problem of representation of knowledge and concepts by using a network or a graph. The basic unit of semantic network is an entity or a concept. A concept could be a tangible or an abstract item like an idea. Sets of concepts have some relation to each other and can be represented with directed or undirected edges. Each edge denoted a specific type of relationship between two concepts.

Let's assume we are talking about the concept of fish. We can have different concepts around fish based on their relationship to it. For instance, fish "is-a" animal and fish "is-a" part of marine life. These relationships are depicted as "is-a" relationships. There can be various other relationships like "has-a," "part-of," "related-to," and many more depending on the context and semantics. These concepts and relationships form a semantic network and you can even browse several of these semantic models online, where you'll find vast knowledgebases spanning different concepts. Figure 1-21 shows a possible representation for concepts related to fish. This model is provided courtesy of Nodebox at `https://www.nodebox.net/perception/`. You can search for various concepts and see associated concepts at this site.



IS-A	HAS-SPECIFIC	IS-PART-OF	HAS-PART	HAS-PROPERTY	IS-RELATED-TO
animal	eel	ocean	fin	wet	heron
marine	piranha	school			
	salmon	sea			
	shark	water			
	shoal				
	tuna				

Figure 1-21. *Semantic network around the concept of fish*

From the network shown in Figure 1-21, we can see some of the concepts we discussed earlier around fish, as well as specific types of fish, like eel, salmon, shark, etc. These can be hyponyms to the concept "fish". These semantic networks are formally denoted and represented by semantic data models using graph structures, where concepts or entities are the nodes and the edges denote the relationships. The semantic web is the extension of the world wide web using semantic metadata annotations and embeddings and data modeling techniques like Resource Description Framework (RDF) and Web Ontology Language (OWL). In linguistics, we have a rich lexical corpus and database called WordNet, which has an exhaustive list of different lexical entities that are grouped into synsets based on semantic similarity (e.g., synonyms). Semantic relationships between these synsets and consequently various words can be explored in WordNet, making it in essence a type of semantic network. We talk about WordNet in more detail in a later section when we cover text corpora.

Representation of Semantics

So far we have seen how to represent semantics based on lexical units and how they can be interconnected by leveraging semantic networks. If we consider the normal form of communication via messages, whether it is written or spoken, if an entity sends a message to another entity and that entity takes some specific actions based on the message, then he/she is said to have understood the meaning conveyed by that message.

A question that might come to mind is how we formally represent the meaning or semantics conveyed by a simple sentence. While it might be extremely easy for us to understand the meaning conveyed, representing semantics formally is not as easy as it seems. Consider the following example, "Get me the book from the table". This sentence by nature is a directive and it directs the listener to do something. Understanding the meaning conveyed by this sentence may involve pragmatics like "which specific book?" and "which specific table?" besides the actual deed of getting the book from the table. While the human mind is intuitive, representing the meaning and relationship between the various constituents formally is a challenge. However, we can do it using several techniques, like propositional logic and first order logic. Using these representations, we can represent the meaning indicated by different sentences and draw inference from them. We can even discover if one sentence entails another one based on their semantics. Representation of semantics is useful especially for carrying out various natural language processing operations in order to make machines understand the semantics behind messages using proper representations, since they lack the cognitive power of humans.

Propositional Logic

Propositional logic, also known as sentential logic or statement logic, is defined as the discipline of logic that's concerned with the study of propositions, statements, and sentences. This includes studying logical relationships and properties between propositions and statements, combining multiple propositions to form more complex propositions, and observing how the value of propositions change based on the components and logical operators. A proposition or statement is usually declarative and is capable of having a binary truth value (true or false). Usually statement is more language specific and concrete and a proposition is more inclined toward the idea or the concepts conveyed by the statement. A simple example is these two statements—"The rocket was faster than the airship" and "The airship was slower than the rocket"—which are distinct but convey the same meaning or proposition. However, the terms statement and proposition are often used interchangeably in propositional logic.

The main focus in propositional logic is to study different propositions and see how combining various propositions with logical operators changes the semantics of the overall proposition. These logic operators are used more like connectors or coordinating conjunctions. Operators include terms like "and," "or," and "not," which can change the meaning of a proposition by itself or when combined with several propositions. A simple example is two propositions—"The Earth is round" and "The Earth revolves around the Sun". These can be combined with the logical operator "and" to give us the proposition, "The Earth is round *and* it revolves around the Sun," which gives us the indication that the two propositions on either side of the "and" operator must be true for the combined proposition to be true.

The good thing about propositional logic is that each proposition has its own truth value and it is not concerned with further subdividing a proposition into smaller chunks and verifying the logical characteristics. Each proposition is considered an indivisible whole unit with its own truth value. Logical operators may be applied to it and several other propositions. Subdividing parts of propositions like clauses or phrases are not considered here. To represent the various building blocks of propositional logic, we use several conventions and symbols. Uppercase letters like **P** and **Q** are used to denote individual statements or propositions. The different operators used and their corresponding symbols are depicted in Table 1-2, based on their order of precedence.

Table 1-2. *Logical Operators with Their Symbols and Precedence*

SI No.	Operator Symbol	Operator Meaning	Precedence
1	¬	not	Highest
2	∧	and	
3	∨	or	
4	→	if-then	
5	↔	iff (if and only if)	Lowest

From Table 1-2, we can see that there are a total of five operators with the not operator having the highest precedence and the iff operator having the lowest. Logical constants are denoted as being true or false. Constants and symbols are known as atomic units and all other units, more specifically the sentences and statements, are complex units. A literal is usually an atomic statement or its negation on applying the *not* operator. We depict a simple example of two sentences, **P** and **Q**, and applying the various operators to them. Let's consider the following representations:

P: He is hungry

Q: He will eat a sandwich

The expression **P ∧ Q** translates to "he is hungry and he will eat a sandwich". This expresses that the outcome of this operation itself is also a sentence or proposition. This is the conjunction operation where **P** and **Q** are the conjuncts. The outcome of this sentence is true only if both **P** and **Q** are true.

The expression **P ∨ Q** translates to "he is hungry or he will eat a sandwich". This expresses that the outcome of this operation is also another proposition formed from the disjunction operation where **P** and **Q** are the disjuncts. The outcome of this sentence is true if either **P** or **Q** is true or both of them are true.

The expression **P → Q** translates to "if he is hungry, then he will eat a sandwich". This is the implication operation that determines **P** is the premise or the antecedent and **Q** is the consequent. It is just like a rule that states that **Q** will occur only if **P** has already occurred or is true.

The expression **P ↔ Q** translates to "he will eat a sandwich if and only if he is hungry," which is basically a combination of the expressions "If he is hungry then he will eat a sandwich" (**P → Q**) and "If he will eat a sandwich, he is hungry" (**Q → P**). This is the biconditional or equivalence operation that will evaluate to true if and only if the two implication operations we described evaluate to true.

The expression **¬P** translates to "he is not hungry," which depicts the negation operation and will evaluate to true if and only if **P** evaluates to false.

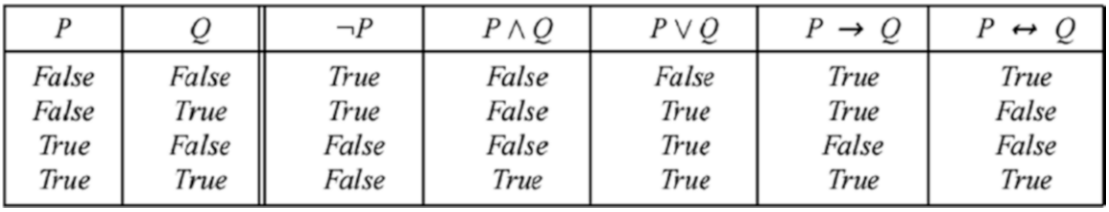
This gives us an idea of the basic operations between propositions and more complex operations, which can be carried out with multiple logical connectives and by adding more propositions. A simple example are these statements:

P: We will play football

Q: The stadium is open

R: It will rain today

They can be combined and represented as **Q ∧ ¬R → P** to depict the complex proposition, "If the stadium is open and it does not rain today, then we will play football". The semantics of the truth value or outcome of the final proposition can be evaluated based on the truth value of the individual propositions and the operators. The various outcomes of the truth values for the different operators are depicted in Figure 1-22.



P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

Figure 1-22. *Truth values for various logical connectors*

Thus, using the table in Figure 1-22, we can evaluate even more complex propositions by breaking them down into simpler binary operations, evaluating the truth value for them, and combining them step by step. Besides these outcomes, there are other properties like associativity, commutativity, and distributivity, which aid in evaluating complex proposition outcomes. The act of checking the validity of each operation and proposition and finally evaluating the outcome is also known as *inference*.

However, besides evaluating extensive truth tables all the time, we can also use several inference rules to arrive at the final outcome or conclusion. The main reason for doing so would be that the size of these truth tables with the various operations starts increasing exponentially as the number of propositions increases. Moreover, rules of inference are easier to understand and well tested and at the heart of them, the same truth value tables are actually applied but we do not have to bother ourselves with the internals. A sequence of inference rules, when applied, usually leads to a conclusion, which is often called a *logical proof*. The usual form of an inference rule is $\mathbf{P} \vdash \mathbf{Q}$, which indicates that **Q** can be derived by some inference operations from the set of statements represented by **P**. The turnstile symbol ($\vdash$) indicates that **Q** is some logical consequence of **P**. The most popular inference rules are as follows:

- **Modus ponens:** This is perhaps the most popular inference rule, also known as the implication elimination rule. It can be represented as $\{\mathbf{P} \rightarrow \mathbf{Q}, \mathbf{P}\} \vdash \mathbf{Q}$, which indicates that if **P** implies **Q** and **P** is asserted to be true, then it is inferred that **Q** is also true. You can also represent this using the following representation $((\mathbf{P} \rightarrow \mathbf{Q}) \wedge \mathbf{P}) \rightarrow \mathbf{Q}$, which can be evaluated easily using truth tables. A simple example is the statement, "If it is sunny, we will play football" represented by $\mathbf{P} \rightarrow \mathbf{Q}$. Now if we say that "it is sunny," this indicates that **P** is true, hence **Q** automatically is inferred as true as well, indicating, "we will play football".

- **Modus tollens:** This is quite similar to the previous rule and is represented formally as **{P → Q, ¬Q}** ⊢ **¬P**. This indicates that if **P** implies **Q** and **Q** is actually asserted to be false, then it is inferred that **P** is false as well. You can also represent this using the following representation (**(P → Q) ∧ ¬Q) → ¬P**, which can be evaluated easily using truth tables. An example proposition is, "If he is a bachelor, he is not married" indicated by **P → Q**. Now if we propose that "he is married," represented by **¬Q**, then we can infer **¬P**, which translates to "he is not a bachelor".

- **Disjunctive syllogism:** This is also known as disjunction elimination and is formally represented as **{P ∨ Q, ¬P}** ⊢ **Q**. This indicates that if either **P** or **Q** is true and **P** is false then **Q** is true. A simple example is the statement, "He is a miracle worker or a fraud" represented by **P ∨ Q**. The statement "he is not a miracle worker" is represented by **¬P**, so we can then infer "he is a fraud," depicted by **Q**.

- **Hypothetical syllogism:** This is often known as the chain rule of deduction and is formally represented as **{P → Q, Q → R}** ⊢ **P → R**. This tells us that if **P** implies **Q** and **Q** implies **R**, we can infer that **P** implies **R**. A really interesting example to understand this is the statement "If I am sick, I can't go to work" represented by **P → Q** and "If I can't go to work, the building construction will not be complete" represented by **Q → R**. We can then infer "If I am sick, the building construction will not be complete," which is represented by **P → R**.

- **Constructive dilemma:** This inference rule is the disjunctive version of modus ponens and can be formally represented as **{(P → Q) ∧ (R → S), P ∨ R}** ⊢ **Q ∨ S**. This indicates that if **P** implies **Q** and **R** implies **S**, and either **P** or **R** are true, then it can be inferred that either **Q** or **S** is true. Consider the following propositions, "If I work hard, I will be successful" represented by **P → Q**, and "If I win the lottery, I will be rich" represented by **R → S**. Now we can propose that, "I work hard *or* I win the lottery" is true, which is represented by **P ∨ R**. We can then infer that "I will be successful or I will be rich" represented by **Q ∨ S**. The complement of this rule is *destructive dilemma,* which is the disjunctive version of *modus tollens*.

This should give you a clear idea of how intuitive inference rules can be and using them is much easier than going over multiple truth tables trying to determine the outcome of complex propositions. The interpretation we derive from inference gives us the semantics of the statement or proposition. A valid statement is one that would be true under all interpretations, irrespective of the logical operations or various statements inside it. This is often called a *tautology*. The complement of a tautology is a *contradiction* or an inconsistent statement, which is false under all interpretations. Note that the previous list is just an indicative list of the most popular inference rules and it is by no way exhaustive. Interested readers can read more on inference and propositional calculus to get an idea of several other rules and axioms, which are used besides the ones covered here to gain more depth into the subject. Next we look at first order logic, which tries to solve some of the shortcomings existing in propositional logic.

First Order Logic

First order logic (FOL), also known popularly as predicate logic and first order predicate calculus, is defined as a collection of well-defined formal systems used extensively in deduction, inference, and representation of knowledge. FOL allows us to use quantifiers and variables in sentences, which enable us to overcome some of the limitations of propositional logic. If we are to consider the pros and cons of propositional logic (PL), considering the points in its favor, PL is declarative and allows us to easily represent facts using a well-formed syntax. PL also allows complex representations like conjunctive, disjunctive, and negated knowledge representations. This by nature makes PL compositional, wherein a composite or complex proposition is built from the simple propositions that are its components along with logical connectives. However, there are several areas where PL is lacking. It is definitely not easy to represent facts in PL because for each possible atomic fact, we need a unique symbolic representation. Hence, due to this limitation, PL has very limited expressive power. Hence the basic idea behind FOL is to not treat propositions as atomic entities.

FOL has a much richer syntax and necessary components for the same compared to PL. The basic components in FOL are as follows:

- **Objects:** These are specific entities or terms with individual unique identities like people, animals, etc.
- **Relations:** These are also known as predicates and usually hold among objects or sets of objects and express some form of relationship or connection like `is_man`, `is_brother`, and `is_mortal`. Relations typically correspond to verbs.
- **Functions:** These are a subset of relations where there is always only one output value or object for some given input. Examples include `height`, `weight`, and `age_of`.
- **Properties:** These are specific attributes of objects that help distinguish them from other objects like round, huge, etc.
- **Connectives:** These are the logical connectives, which are similar to the ones in PL, which include not (¬), and (∧), or (∨), implies (→), and iff (if and only if ↔).
- **Quantifiers:** These include two types of quantifiers—universal (∀) which stands for "for all" or "all" and existential (∃), which stands for "there exists" or "exists". They are used for quantifying entities in a logical or mathematical expression.
- **Constant symbols:** These are used to represent concrete entities or objects in the world. Examples include John, King, Red, and 7.
- **Variable symbols:** These are used to represent variables like x, y, and z.
- **Function symbols:** These are used to map functions to outcomes, Examples include `age_of(John)` = 25 or `color_of(Tree)` = Green.
- **Predicate symbols:** These map specific entities and a relation or function between them to a truth value based on the outcome. Examples include `color(sky, blue)` = True.

These are the main components that go into logical representations and syntax for FOL. Usually, objects are represented by various terms, which could be a function, variable, or a constant based on the different components. These terms do not need to be defined and do not return values. Various propositions are usually constructed using predicates and terms with the help of predicate symbols. An n-ary predicate is constructed from a function over n-terms, which have either a true or false outcome. An atomic sentence can be represented by an n-ary predicate and the outcome is true or false depending on the semantics of the sentence, i.e., if the objects represented by the terms have the correct relation as specified by the predicate. A *complex sentence* or statement is formed using several atomic sentences and logical connectives. A *quantified sentence* adds the quantifiers mentioned earlier to sentences.

Quantifiers are one of the advantages FOL has over PL, since they enable us to represent statements about entire sets of objects without needing to represent and enumerate each object by a different name. The *universal quantifier* (∀) asserts that a specific relation or predicate is true for all values associated with a specific variable. The representation **∀x F(x)** indicates that F holds for all values of x in the domain associated with x. An example is ∀x cat(x) → animal(x), which indicates that all cats are animals. Universal quantifiers are used with the *implies* (→) connective to form rules and statements. An important point to remember is that universal quantifiers are almost never used in statements to indicate some relation for every entity in the world using the conjunction (∧) connective. An example would be the representation, ∀x dog(x) ∧ eats_meat(x), which actually means that every entity in the world is a dog and they eat meat. This obviously sounds absurd! The *existential quantifier* (∃) asserts that a specific relation or predicate holds true for at least some value associated with a specific variable. The representation, **∃x F(x)** indicates that F holds for some value of x in the domain associated with x. An example is ∃x student(x) ∧ pass_exam(x), which indicates that there is at least one student who has passed the exam. This quantifier gives FOL a lot of power since we can make statements about objects or entities without specifically naming them.

Existential quantifiers are usually used with the *conjunction* (∧) connective to form rules and statements. You should remember that existential quantifiers are almost never used with the *implies* (→) connective in statements because the semantics indicated by it are usually wrong. An example is ∃x student(x) → knowledgeable(x) which tells us if you are a student you are knowledgeable. The real problem happens if you ask, what about those who are not students, are they not knowledgeable?

Considering the scope for nesting of quantifiers, ordering of multiple quantifiers may or may not matter depending on the type of quantifiers used. For multiple universal quantifiers, switching the order does not change the meaning of the statement. This can be depicted by (∀x)(∀y) brother(x,y) ↔ (∀y)(∀x) brother(x,y) which indicates two people are brothers, irrespective of the order. Similarly, you can also switch the order of existential quantifiers like (∃x)(∃y) F(x,y) ↔ (∃y)(∃x) F(x,y), Switching the order of mixed quantifiers in a sentence does matter and changes the interpretation of that sentence. This can be explained more clearly in the following examples, which are very popular in FOL.

- (∀x)(∃y) loves(x, y) means that everyone in the world loves at least someone.
- (∃y)(∀x) loves(x, y) means that someone is the world is loved by everyone.
- (∀y)(∃x) loves(x, y) means that everyone in the world has at least someone who loves them.
- (∃x)(∀y) loves(x, y) means that there is at least someone in the world who loves everyone.

From these examples, you can see how the statements look almost the same but the ordering of quantifiers changes the meanings significantly. There are also several other properties showing the relationship between the quantifiers. We list some of the popular quantifier identities and properties as follows.

- (∀x) ¬F(x) ↔ ¬(∃x) F(x)
- ¬(∀x) F(x) ↔ (∃x) ¬F(x)
- (∀x) F(x) ↔ ¬ (∃x) ¬F(x)
- (∃x) F(x) ↔ ¬(∀x) ¬F(x)
- (∀x) (P(x) ∧ Q(x)) ↔ ∀x P(x) ∧ ∀x Q(x)
- (∃x) (P(x) ∨ Q(x)) ↔ ∃x P(x) ∨ ∃x Q(x)

There are a couple of other important concepts for transformation rules in predicate logic. These include instantiation and generalization. Universal instantiation, also known as *universal elimination,* is a rule of inference involving the universal quantifier. It tells us that if (∀x) F(x) is true, then F(C) is true where C is any constant term that is present in the domain of x. The variable symbol here can be replaced by any ground term. An example depicting this would be (∀x) `drinks(John, x)` → `drinks(John, Water)`.

Universal generalization, also known as *universal introduction,* is the inference rule that tells us that if F(A) ∧ F(B) ∧ F(C) ∧ … so on hold true, then we can infer that (∀x) F(x) holds true.

Existential instantiation, also known as *existential elimination,* is an inference rule involving the existential quantifier. It tells us that if the given representation (∃x) F(x) exists, we can infer F(C) for a new constant or variable symbol C. This is assuming that the constant or variable term C introduced in this rule is a new constant that has not occurred previously in this proof or in our existing knowledge base. This process is also known as *skolemization* and the constant C is known as the skolem constant.

Existential generalization, also known as *existential introduction,* is the inference rule that tells us that assuming F(C) to be true where C is a constant term, we can then infer (∃x) F(x) from it. This can be depicted by the representation, `eats_fish(Cat)` → (∃x) `eats_fish(x)`, which can be translated as "Cats eat fish, and therefore there exists something or someone at least who eats fish".

We now look at some examples of how FOL is used to represent natural language statements and vice versa. The examples in Table 1-3 depict the typical usage of FOL to represent natural language statements.

Table 1-3. *Representation of Natural Language Statements Using First Order Logic*

SI No.	FOL Representation	Natural Language Statement
1	¬ `eats(John, fish)`	John does not eat fish.
2	`is_hot(pie)` ∧ `is_delicious(pie)`	The pie is hot and delicious.
3	`is_hot(pie)` ∨ `is_delicious(pie)`	The pie is either hot or delicious.
4	`study(John, exam)` → `pass(John, exam)`	If John studies for the exam, he will pass the exam.
5	∀x `student(x)` → `pass(x, exam)`	All students passed the exam.
6	∃x `student(x)` ∧ `fail(x, exam)`	There is at least one student who failed the exam.
7	(∃x `student(x)` ∧ `fail(x, exam)` ∧ (∀y `fail(y, exam)` → x=y))	There was exactly one student who failed the exam.
8	∀x `(spider(x)` ∧ `black_widow(x))` → `poisonous(x)`	All black widow spiders are poisonous.

This gives us a good idea about the various components of FOL and the utility and advantages it gives us over propositional logic. However, FOL has its own limitations. By nature, it allows us to quantify over variables and objects, but not properties or relations. Higher order logic (HOL) allows us to quantify over relations, predicates, and functions. More specifically, second order logic enables us to quantify over predicates and functions and third order logic enables us to quantify over predicates of predicates. While they are more expressive, it is extremely difficult to determine the validity of all sentences in HOL. This brings us to an end of our discussion on representing semantics. We talk about text corpora in the following section.

Text Corpora

Text corpora is the plural form of "text corpus" and can be defined as large and structured collections of texts or textual data. They usually consist of a body of written or spoken text, often stored in electronic form. This includes converting old historic text corpora from physical to electronic form so that they can be analyzed and processed with ease. The primary purpose of text corpora is to leverage them for linguistic as well as statistical analysis and to use them as data for building natural language processing tools.

Monolingual corpora consist of textual data in only one language and multilingual corpora consist of textual data in multiple languages. To understand the significance of text corpora, we must understand the origins of corpora and the reason behind them. It all started with the emergence of linguistics and people collecting data related to language to study its properties and structure. During the 1950s, statistical and quantitative methods were used to analyze collected data. However, this soon reached a dead end due to the lack of large amounts of textual data over which statistical methods could be effectively applied. Besides this, cognitive learning and behavioral sciences gained a lot of focus. This empowered eminent linguist Noam Chomsky to build and formulate a sophisticated rule-based language model that formed the basis for building, annotating, and analyzing large-scale text corpora.

Corpora Annotation and Utilities

Text corpora are annotated with rich metadata that is extremely useful for getting valuable insights when utilizing the corpora for natural language processing and text analytics. Popular annotations for text corpora include tagging parts of speech or POS tags, word stems, lemmas, and many more. We look at some of the most commonly used methods and techniques for annotating text corpora.

- **POS tagging:** This is mainly used to annotate each word with a POS tag indicating the part of speech associated with it.
- **Word stems:** A stem for a word is a part of the word to which various affixes can be attached.
- **Word lemmas:** A lemma is the canonical or base form for a set of words and is also known as the head word.
- **Dependency grammar:** These include finding out the various relationships among the components in sentences and annotating the dependencies.
- **Constituency grammar:** These are used to add syntactic annotation to sentences based on their constituents, including phrases and clauses.
- **Semantic types and roles:** The various constituents of sentences, including words and phrases, are annotated with specific semantic types and roles often obtained from an ontology that indicates what they do. These include things like place, person, time, organization, agent, recipient, theme, etc.

Advanced forms of annotations include adding syntactic and semantic structure for text. These are dependency and constituency grammar-based parse trees. These specialized corpora are also known as *treebanks*, which are extensively used in building POS taggers, syntax, and semantic parsers. Corpora are also used extensively by linguists to create new dictionaries and grammar rules. Properties like concordance, collocations, and frequency counts enable them to find lexical information, patterns, morphosyntactic information, and language learning. Besides linguistics, corpora are widely used in developing natural language processing tools like text taggers, speech recognition, machine translation, spelling and grammar checkers, text-to-speech and speech-to-text synthesizers, information retrieval, entity recognition, and knowledge extraction.

Popular Corpora

There are several popular resources for text corpora that have been built and have evolved over time. We list some of the most popular corpora in this section to whet your appetite and you can further go ahead and find out more details about the text corpora that catch your eye. The following list describes some popular text corpora built over time.

- **Keyword in context:** KWIC was a methodology invented in the 1860s but used extensively around the 1950s by linguists to index documents and create corpora of concordances.
- **Brown corpus:** This was the first million-word corpus for the English language, published by Kucera and Francis in 1961; it's also known as "A Standard Corpus of Present-Day American English". This corpus consists of text from a wide variety of sources and categories.
- **LOB corpus:** The Lancaster-Oslo-Bergen (LOB) corpus was compiled in the 1970s as a result of collaboration between the University of Lancaster, the University of Oslo, and the Norwegian Computing Centre for the Humanities, Bergen. The main motivation of this project was to provide a British counterpart to the Brown corpus. This corpus is also a million-word corpus consisting of text from a wide variety of sources and categories.
- **Collins corpus:** The Collins Birmingham University International Language Database (COBUILD) set up in 1980 in the University of Birmingham and funded by the Collins publishers built a large electronic corpus of contemporary text in the English language and paved the way for future corpora like the "Bank of English" and the "Collins COBUILD English Language Dictionary".
- **CHILDES:** The Child Language Data Exchange System (CHILDES) is a corpus that was created by Brian and Catherine in 1984 and serves as a repository for language acquisition data including transcripts, audio, and video in 26 languages from over 130 different corpora. This was recently merged with a larger corpus called Talkbank. It is used extensively for analyzing the language and speech of young children.

- **WordNet:** This corpus is a semantic oriented lexical database for the English language. It was created at Princeton University in 1985 under the supervision of George Armitage. The corpus consists of words and synonym sets often termed *synsets*. Besides these, it consists of word definitions, relationships, and examples of using words and synsets. Overall, it is a combination of a dictionary as well as a thesaurus.
- **Penn treebank:** This corpus consists of tagged and parsed English sentences including annotations like POS tags and grammar-based parse trees typically found in treebanks. It can be also defined as a bank of linguistic trees and was created at the University of Pennsylvania.
- **BNC corpus:** The British National Corpus (BNC) is one of the largest English corpora, consisting of over 100 million written and spoken text samples from a wide variety of sources. This corpus is a representative sample of written and spoken British English of the late 20th Century.
- **ANC corpus:** The American National Corpus (ANC) is a large text corpus in American English. It consists of over 22 million spoken and written text samples since the 1990s. It includes data from a wide variety of sources, including some emerging sources like email, tweets, and websites, which are not present in the BNC.
- **COCA corpus:** The Corpus of Contemporary American English (COCA) is the largest text corpus in American English and consists of over 450 million words, including spoken transcripts and written text from various categories and sources.
- **Google N-gram corpus:** The Google N-gram corpus consists of over a trillion words from various sources, including books, web pages, and so on. The corpora consists of n-gram files up to 5-grams for each language.
- **Reuters corpus:** This corpus is a collection of Reuters news articles and stories released in 2000 specifically for carrying out research in natural language processing and machine learning.

- **Web, chat, email, tweets:** These are entirely new forms of text corpora that have come into prominence since the rise of social media. We can get them on the web from various sources, including Twitter, Facebook, chat rooms, and so on.

This gives us an idea of some of the most popular text corpora and how they have evolved over time. In the next section, we talk about how we can access some of these text corpora with the help of Python and the Natural Language Toolkit (NLTK) platform.

Accessing Text Corpora

We already have an idea about what constitutes a text corpus and looked at a list of several popular text corpora that exist today. In this section, we leverage NLTK to interface and access some of these text corpora. We cover NLTK and Python more in the next chapter, so do not worry if some of the syntax or code seems to be a bit overwhelming. The main intent of this section is to give you an idea of how you can access and utilize text corpora easily for your natural language processing and analytics needs. We also use the Natural Language Toolkit (`nltk`) library and you can find out more details about this project at http://www.nltk.org/, which tells us more about NLTK being a complete platform and framework for accessing text resources, including corpora and libraries for various natural language processing and machine learning capabilities.

To start, make sure you have Python installed. You can install Python separately or download the popular Anaconda Python distribution from Anaconda at https://www.anaconda.com/download, which comes with a complete suite of analytics packages, including NLTK. If you want more details about Python and want to determine which distribution would be best suited for you, you can go to Chapter 2 and take a quick glance where we cover these topics in further detail.

Assuming you have Python installed now, if you installed the Anaconda distribution, you will already have NLTK installed. Note that we use Python 3 for the entire course of this book and we recommend everyone use the latest version of Python, preferably at least Python 3.5+. In case you did not install the Anaconda distribution but have Python installed, you can open your terminal or command prompt and run the following command to install NLTK.

```
$ pip install nltk
```

This will install the nltk library and you will be ready to use it. However, the default installation of NLTK does not include all the components required in this book. To install all the components and resources of NLTK, you can start your Python shell and type the following commands. You will see the various dependencies for NLTK being downloaded; a part of the output is shown in the following code snippet.

```
In [1]: import nltk

In [2]: nltk.download('all', halt_on_error=False)
[nltk_data] Downloading collection u'all'
[nltk_data]    |
[nltk_data]    | Downloading package abc to
[nltk_data]    |     C:\Users\DIP.DIPSLAPTOP\AppData\Roaming\nltk_data
[nltk_data]    |     ...
[nltk_data]    |   Package abc is already up-to-date!
[nltk_data]    | Downloading package alpino to
[nltk_data]    |     C:\Users\DIP.DIPSLAPTOP\AppData\Roaming\nltk_data
[nltk_data]    |     ...
```

This command will download all the resources required by NLTK. In case you do not want to download everything, you can also select the necessary components from a graphical user interface (GUI) using the nltk.download() command. Once the necessary dependencies are downloaded, you are now ready to start accessing the text corpora!

Accessing the Brown Corpus

We have already talked a bit about the Brown corpus, which was developed in 1961 at Brown University. This corpus consists of texts from 500 sources and has been grouped into various categories. The following code snippet loads the Brown corpus into the system memory and shows the various categories.

```
# load the Brown Corpus
from nltk.corpus import brown

# total categories
print('Total Categories:', len(brown.categories()))
Total Categories: 15
```

```
# print the categories
print(brown.categories())
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government',
'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion',
'reviews', 'romance', 'science_fiction']
```

The preceding output tells us that there are a total of 15 categories in the corpus, including news, mystery, lore, and so on. The following code snippet digs a little deeper into the mystery category of the Brown corpus.

```
# tokenized sentences
brown.sents(categories='mystery')
[['There', 'were', 'thirty-eight', 'patients', 'on', 'the', 'bus', 'the',
'morning', 'I', 'left', 'for', 'Hanover', ',', 'most', 'of', 'them',
'disturbed', 'and', 'hallucinating', '.'], ['An', 'interne', ',', 'a',
'nurse', 'and', 'two', 'attendants', 'were', 'in', 'charge', 'of', 'us',
'.'], ...]
```

```
# POS tagged sentences
brown.tagged_sents(categories='mystery')
[[('There', 'EX'), ('were', 'BED'), ('thirty-eight', 'CD'), ('patients',
'NNS'), ('on', 'IN'), ('the', 'AT'), ('bus', 'NN'), ('the', 'AT'),
('morning', 'NN'), ('I', 'PPSS'), ('left', 'VBD'), ('for', 'IN'),
('Hanover', 'NP'), (',', ','), ('most', 'AP'), ('of', 'IN'), ('them',
'PPO'), ('disturbed', 'VBN'), ('and', 'CC'), ('hallucinating', 'VBG'),
('.', '.')], [('An', 'AT'), ('interne', 'NN'), (',', ','), ('a', 'AT'),
('nurse', 'NN'), ('and', 'CC'), ('two', 'CD'), ('attendants', 'NNS'),
('were', 'BED'), ('in', 'IN'), ('charge', 'NN'), ('of', 'IN'), ('us',
'PPO'), ('.', '.')], ...]
```

```
# get sentences in natural form
sentences = brown.sents(categories='mystery')
sentences = [' '.join(sentence_token) for sentence_token in sentences]
sentences[0:5] # viewing the first 5 sentences
['There were thirty-eight patients on the bus the morning I left for
Hanover , most of them disturbed and hallucinating .',
 'An interne , a nurse and two attendants were in charge of us .',
 "I felt lonely and depressed as I stared out the bus window at Chicago's
grim, dirty West Side.",
 'It seemed incredible , as I listened to the monotonous drone of voices
and smelled the fetid odors coming from the patients , that technically I
was a ward of the state of Illinois , going to a hospital for the mentally
ill .',
 'I suddenly thought of Mary Jane Brennan , the way her pretty eyes could
flash with anger , her quiet competence , the gentleness and sweetness that
lay just beneath the surface of her defenses .']
```

From the preceding snippet and output, we can see the written contents of the mystery genre and see how the sentences are available in tokenized as well as annotated formats. Suppose we want to see the top nouns in the mystery genre? We can use the following code snippet to obtain them. Remember that nouns have either an NN or NP in their POS tag to indicate various forms of nouns. We cover POS tags in further detail in Chapter 3.

```
# get tagged words
tagged_words = brown.tagged_words(categories='mystery')

# get nouns from tagged words
nouns = [(word, tag) for word, tag in tagged_words if any(noun_tag in tag
                                                          for noun_tag in
['NP', 'NN'])]
nouns[0:10] # view the first 10 nouns
[('patients', 'NNS'), ('bus', 'NN'), ('morning', 'NN'), ('Hanover', 'NP'),
('interne', 'NN'),
 ('nurse', 'NN'), ('attendants', 'NNS'), ('charge', 'NN'), ('bus', 'NN'),
('window', 'NN')]

# build frequency distribution for nouns
nouns_freq = nltk.FreqDist([word for word, tag in nouns])

# view top 10 occurring nouns
nouns_freq.most_common(10)
[('man', 106), ('time', 82), ('door', 80), ('car', 69), ('room', 65),
 ('Mr.', 63), ('way', 61), ('office', 50), ('eyes', 48), ('Mrs.', 46)]
```

The preceding snippet and outputs depict the top ten nouns that occur most frequently. It includes terms like man, time, room, and so on. We have used some advanced constructs and techniques like list comprehensions, iterables, and tuples. We cover core concepts of text processing using Python constructs in the next chapter. For now, all you need to know is that we filter out the nouns from all other words based on their POS tags and then compute their frequency to get the top occurring nouns in the corpus.

Accessing the Reuters Corpus

The Reuters corpus consists of 10,788 Reuters news documents from around 90 different categories and has been grouped into train and test sets. In machine learning terminology, train sets are used to train a model and test sets are used to test the performance of that model. The following code snippet shows us how to access the data for the Reuters corpus.

```
# load the Reuters Corpus
from nltk.corpus import reuters

# total categories
print('Total Categories:', len(reuters.categories()))
Total Categories: 90

# print the categories
print(reuters.categories())
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa', ...,
'yen', 'zinc']

# get sentences in housing and income categories
sentences = reuters.sents(categories=['housing', 'income'])
sentences = [' '.join(sentence_tokens) for sentence_tokens in sentences]
sentences[0:5]  # view the first 5 sentences
["YUGOSLAV ECONOMY WORSENED IN 1986 , BANK DATA SHOWS National Bank
economic data for 1986 shows that Yugoslavia ' s trade deficit grew , the
inflation rate rose , wages were sharply higher , the money supply expanded
and the value of the dinar fell .",
 'The trade deficit for 1986 was 2 . 012 billion dlrs , 25 . 7 pct higher
than in 1985 .',
 'The trend continued in the first three months of this year as exports
dropped by 17 . 8 pct , in hard currency terms , to 2 . 124 billion dlrs .',
 'Yugoslavia this year started quoting trade figures in dinars based on
current exchange rates , instead of dollars based on a fixed exchange rate
of 264 . 53 dinars per dollar .',
 "Yugoslavia ' s balance of payments surplus with the convertible currency
area fell to 245 mln dlrs in 1986 from 344 mln in 1985 ."]

# fileid based access
print(reuters.fileids(categories=['housing', 'income']))
['test/16118', 'test/18534', 'test/18540', ..., 'training/7006',
'training/7015', 'training/7036', 'training/7098', 'training/7099',
'training/9615']

print(reuters.sents(fileids=[u'test/16118', u'test/18534']))
[['YUGOSLAV', 'ECONOMY', 'WORSENED', 'IN', '1986', ',', 'BANK', 'DATA',
'SHOWS', 'National', 'Bank', 'economic', 'data', 'for', '1986', 'shows',
'that', 'Yugoslavia', "'", 's', 'trade', 'deficit', 'grew', ',', 'the',
'inflation', 'rate', 'rose', ',', 'wages', 'were', 'sharply', 'higher',
',', 'the', 'money', 'supply', 'expanded', 'and', 'the', 'value', 'of',
'the', 'dinar', 'fell', '.'], ['The', 'trade', 'deficit', 'for', '1986',
'was', '2', '.', '012', 'billion', 'dlrs', ',', '25', '.', '7', 'pct',
'higher', 'than', 'in', '1985', '.'], ...]
```

This gives us an idea of how to access corpora data using both categories as well as file identifiers. Next, we look at how to access the WordNet corpus.

Accessing the WordNet Corpus

The WordNet corpus is perhaps one of the most used corpora out there since it consists of a vast corpus of words and semantically linked synsets for each word. We explore some of the basic features of the WordNet corpus here, including synsets and methods for accessing the corpus data. For more advanced analysis and coverage of WordNet capabilities, you can look at Chapter 3, where we cover synsets, lemmas, hyponyms, hypernyms, and several other concepts that we covered in our semantics section earlier. The following code snippet gives us an idea about how to access the WordNet corpus data and synsets.

```
# load the Wordnet Corpus
from nltk.corpus import wordnet as wn

word = 'hike' # taking hike as our word of interest
# get word synsets
word_synsets = wn.synsets(word)
word_synsets

# get details for each synonym in synset
for synset in word_synsets:
    print(('Synset Name: {name}\n'
           'POS Tag: {tag}\n'
           'Definition: {defn}\n'
           'Examples: {ex}\n').format(name=synset.name(),
                                      tag=synset.pos(),
                                      defn=synset.definition(),
                                      ex=synset.examples()))

Synset Name: hike.n.01
POS Tag: n
Definition: a long walk usually for exercise or pleasure
Examples: ['she enjoys a hike in her spare time']

Synset Name: rise.n.09
POS Tag: n
Definition: an increase in cost
Examples: ['they asked for a 10% rise in rates']

Synset Name: raise.n.01
POS Tag: n
Definition: the amount a salary is increased
Examples: ['he got a 3% raise', 'he got a wage hike']

Synset Name: hike.v.01
POS Tag: v
Definition: increase
Examples: ['The landlord hiked up the rents']
```

```
Synset Name: hike.v.02
POS Tag: v
Definition: walk a long way, as for pleasure or physical exercise
Examples: ['We were hiking in Colorado', 'hike the Rockies']
```

This code snippet depicts an interesting example with the word hike and its synsets, which include synonyms that are nouns as well as verbs having distinct meanings. WordNet makes it easier to semantically link words with synonyms and easily retrieve meanings and examples for various words. This example tells us that "hike" can mean a long walk as well as an increase in prices for salary/rent. Feel free to experiment with different words and determine their synsets, definitions, examples, and relationships.

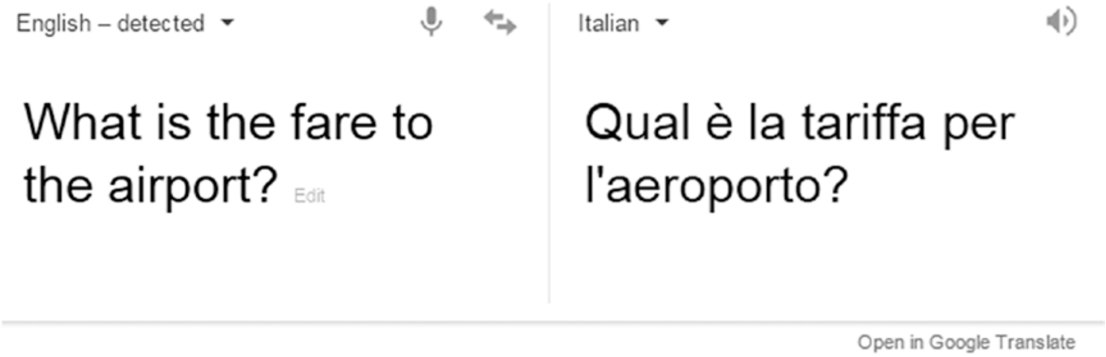
Besides these popular corpora, there are a vast number of text corpora available, which you can check just by looking into the `nltk.corpus` module, which can be used to access any of these corpora. Thus, you can see how easy it is to access and use data from any text corpus with the help of Python and NLTK. This brings us to the end of our discussion about text corpora. In the following section, we cover some ground regarding basic concepts around natural language processing and text analytics.

Natural Language Processing

We mention the term *natural language processing* (NLP) several times in this chapter. By now, you might have formed some idea about what NLP means. NLP is defined as a specialized field of computer science and engineering and artificial intelligence with roots in computational linguistics. It is primarily concerned with designing and building applications and systems that enable interaction between machines and natural languages created by humans. This also makes NLP related to the area of human-computer interaction (HCI). NLP techniques enable computers to process and understand human natural language and utilize it further to provide useful output. Next, we talk about some of the main applications of NLP.

Machine Translation

Machine translation is perhaps one of the most coveted and sought after applications of NLP. It is defined as the technique that helps provide syntactic, grammatical, and semantically correct translations between any two pair of languages. This was perhaps the first major area of research and development in NLP. On a simple level, machine translation is the translation of natural language carried out by a machine. By default, the basic building blocks for the machine translation process involve simple substitution of words from one language to another, but in that case we ignore things like grammar and phrasal structure consistency. Hence, more sophisticated techniques have evolved over time, including combining large resources of text corpora along with statistical and linguistic techniques. One of the most popular machine translation systems is Google Translate. Figure 1-23 shows a successful machine translation operation executed by Google Translate for the sentence, "What is the fare to the airport?" from English to Italian.



What is the fare to the airport?

Qual è la tariffa per l'aeroporto?

Figure 1-23. *Machine translation performed by Google Translate*

Over time, machine translation systems are getting better at providing translations in real time as you speak or write into the application.

Speech Recognition Systems

This is perhaps the most difficult application for NLP. One of the main and perhaps the most difficult tests of true intelligence in artificial intelligence systems is the Turing test. This test states that if a question is given by the user to the computer and to a human, it would be unable to distinguish the responses obtained. Over a period of time, a lot of progress has been made in this area by using techniques like speech synthesis, analysis, syntactic parsing, and contextual reasoning. However, one chief limitation for speech recognition systems still remains that they are very domain-specific and will not work if the user strays even a little bit from the expected scripted inputs needed by the system. Speech recognition systems are now found in a large variety of places from your computers, to mobile phones, to virtual assistance systems.

Question Answering Systems

Question Answering Systems (QAS) are built on the principle of question answering based on using techniques from NLP and information retrieval (IR). QAS is primarily concerned with building robust and scalable systems that provide answers to questions given by users in natural language form. Imagine being in a completely different country, asking a question to your personalized assistant in your phone in pure natural language, and getting a similar response from it. This is the ideal state toward which researchers and technologists are working day in and day out. We have achieved some success in this field with personalized assistants like Siri and Cortana, but their scope is still limited since they understand only a subset of key clauses and phrases in the entire human natural language.

To build a successful QAS, you need a huge knowledgebase consisting of data about various domains. Efficient querying systems into this knowledgebase would be leveraged by the QAS to provide answers to questions in natural language form. Creating and maintaining a queryable vast knowledgebase is extremely difficult, hence you will find the rise of QAS in niche domains like food, healthcare, ecommerce, and so on. Chatbots are one of the emerging trends that extensively use QAS.

Contextual Recognition and Resolution

This covers a wide area in understanding natural language, which includes syntactic and semantic based reasoning. Word sense disambiguation is a popular application where we want to find the contextual sense of a word in a given sentence. Consider the word "book". It can mean an object containing knowledge and information when used as a noun and it can also mean to reserve something like a seat or a table when used as a verb. Detecting these differences in sentences based on context is the main premise of word-sense disambiguation and it is a daunting task.

Co-reference resolution is another problem in linguistics that NLP is trying to address. By definition, co-reference is said to occur when two or more terms/expressions in a body of text refer to the same entity. Then they are said to have the same referent. Consider the example sentence, "John just told me that he is going to the exam hall". In this sentence, the pronoun "he" has the referent "John". Resolving these pronouns is part of co-reference resolution and it becomes challenging once we have multiple referents in a body of text. An example body of text would be, "John just talked with Jim. He told me we have a surprise test tomorrow". In this body of text, the pronoun "he" could refer to either "John" or "Jim", thus making it difficult to pinpoint to the exact referent.

Text Summarization

The main aim of text summarization is to take a corpus of text documents, which could be a collection of texts, paragraphs, or sentences, and reduce the content appropriately to create a summary that retains the key points of the collection of documents. Summarization can be carried out by looking at the various documents and trying to find the keywords, phrases, and sentences that have prominence in the collection. Two main types of techniques for text summarization include extraction-based summarization and abstraction-based summarization. With the advent of huge amounts of text and unstructured data, the need for text summarization for getting to valuable insights quickly is in great demand.

Text summarization systems usually perform two main types of operations. The first one is generic summarization, which tries to provide a generic summary of the collection of documents under analysis. The second type of operation is query-based summarization, which provides query-relevant text summaries where the corpus is filtered further based on specific queries and relevant keywords and phrases are extracted relevant to the query and the summary is constructed.

Text Categorization

The main aim of text categorization is to identify to which category or class a specific document should be placed based on the contents of the document. This is one of the most popular applications of NLP and machine learning because with the right data, it is extremely simple to understand the principles behind its internals and implement a working text categorization system. Both supervised and unsupervised machine learning techniques can be used to solve this problem and sometimes a combination of both are used. This has helped build many successful and practical applications, including spam filters and news article categorizations.

Text Analytics

Like we mentioned before, with the advent of huge amounts of computing power, unstructured data, and success with machine learning and statistical analysis techniques, it wasn't long before text analytics started garnering a lot of attention. However, you need to understand some of the challenges that text analytics poses compared to regular analytical methods. Free-flowing text is highly unstructured and rarely follows a specific pattern, like weather data or structured attributes in relational databases. Hence standard statistical methods will not be helpful when applied out-of-the-box on unstructured text data. In this section, we cover some of the main concepts surrounding text analytics and discuss the definition and scope of text analytics, which will give you a broad idea of what you can expect in the upcoming chapters.

Text analytics, also known as text mining, is defined as the methodology and process followed to derive quality and actionable information and insights from textual data. This involves using natural language processing, information retrieval, and machine learning techniques to parse unstructured text data into more structured forms and deriving patterns and insights from this data that would be helpful to the end user. Text analytics comprises a collection of machine learning, linguistic, and statistical techniques that are used to model and extract information from text primarily for analysis needs, including business intelligence, exploratory, descriptive, and predictive analysis. Some of the main techniques and operations in text analytics are mentioned as follows.

- Text classification
- Text clustering
- Text summarization
- Sentiment analysis
- Entity extraction and recognition
- Similarity analysis and relation modeling

However, doing text analytics is a more involved process sometimes compared to normal statistical analysis or machine learning. The reason is that before applying a learning technique or algorithm, we have to convert the unstructured text data into a format acceptable to those algorithms. By definition, a body of text under analysis is often termed a document and by applying various techniques, we convert this to a vector of words. This is usually a numeric array whose values are specific weights for each word, which could be its frequency, its occurrence, or various other depictions, some of which we explore in Chapter 3. Often the text needs to be cleaned and processed to remove noisy terms and data and this process is termed text preprocessing. Once we have the data in a machine readable and understandable format, we can apply relevant algorithms based on the problem to be solved. The applications of text analytics are manifold and some popular ones are as follows.

- Spam detection
- News articles categorization
- Social media analysis and monitoring
- Biomedical
- Security intelligence
- Marketing and CRM
- Sentiment analysis
- Ad placements
- Chatbots
- Virtual assistants

Machine Learning

We can define machine learning (ML) as a subfield of artificial intelligence (AI). Machine learning is the art and science of leveraging techniques. It can allow machines to automatically learn latent patterns and relationships from underlying data and improve itself over time, without explicitly programming or hard-coding specific rules. Usually a combination of NLP and ML is often needed to solve real-world problems like text categorization, clustering, and so on. The three major categories of machine learning techniques include supervised, unsupervised, and reinforcement learning algorithms.

Deep Learning

The field of deep learning (DL) is a subfield of machine learning specializing in models and algorithms, which have been inspired by how the brain works and functions. Indeed the artificial neural network (ANN) was the first model built by drawing inspiration from the human brain. Although we are definitely quite far away from replicating what the brain does, neural networks are extremely complex non-linear models, which are capable of automatically learning hierarchical data representations. Deep learning or deep neural networks typically use multiple layers of non-linear processing units, also known as neurons. We prefer calling them processing units. Each layer performs some feature extraction, engineering, and transformation on its own using the output from the previous layer as its input. Hence, each level ends up learning hierarchical representations of the data at different levels of abstraction. We can use these models to solve both supervised and unsupervised problems. Recently, deep learning has shown a lot of promise with regard to solving NLP problems.

Summary

Congratulations on staying with this long chapter! We have started our journey of text analytics with Python by taking a journey in the world of natural language processing and the various concepts and domains surrounding it. You now have a good idea of what natural language means and how it is significant in our world. We have also seen concepts surrounding the philosophy of language and language acquisition and usage. The field of linguistics gave us a flavor of the origins of language studies and how they have evolved with time. We covered language syntax and semantics in detail, including the essential concepts with interesting hands-on examples in Python to easily understand them. We also talked about resources for natural language, namely text corpora, and looked at some practical examples with code regarding how to interface and access corpora using Python and NLTK. Finally, we ended with a discussion about the various facets of natural language processing and text analytics. In the next chapter, we talk about using Python for text analytics, where we touch upon setting up your Python development environment for natural language processing, the various constructs of Python useful for text processing, and look at some of the popular state-of-the-art libraries, frameworks, and platforms used for NLP.

CHAPTER 2

Python for Natural Language Processing

In the previous chapter, we took a journey into the world of natural language processing and explored several interesting concepts and domains associated with it. We now have a better understanding of the entire scope surrounding natural language processing, linguistics, and text analytics. If you refresh your memory, we also got our first taste of running Python code to look at essentials with regard to processing and understanding text. We also looked at ways to access and use text corpora resources with the help of the NLTK framework. In this chapter, we look at why Python is the language of choice for natural language processing (NLP), set up a robust Python environment, take a hands-on based approach to understanding essentials of string and text processing, manipulation, and transformation, and conclude by looking at some of the important libraries and frameworks associated with NLP and text analytics. This chapter is aimed to provide a quick refresher for getting started with Python and NLP.

This book assumes you have some knowledge of Python or any other programming language. If you are a Python practitioner or even a Python guru, you can skim through the chapter since a lot of the content here will start right from a brief history of Python, setting up your Python development environment, to basics of Python for working with text data. Our main focus in the chapter is exploring how text data is handled in Python and learning more about the state-of-the-art NLP tools and frameworks in Python. This chapter follows a more hands-on approach and we cover various concepts with practical examples. All the code examples showcased in this chapter are available on the book's official GitHub repository at `https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition`.

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_2

Getting to Know Python

Before we can dive into the Python ecosystem and look at the various components associated with it, we need to take a brief look at the origins and philosophy behind Python and see how it has evolved over time to be the choice of language powering many applications, servers, and systems today. Python is a high-level open source general-purpose programming language widely used as a scripting as well as a programming language across different domains. Python is the brainchild of Guido Van Rossum and was conceived in the late 1980s as a successor to the ABC language, both of which were developed at the Centrum Wiskunde and Informatica (CWI), Netherlands. Python was originally designed to be a scripting and interpreted language and to this day it retains the essence of being one of the most popular scripting languages out there. But with object oriented principles (OOP) and constructs, you can use it just like any other object oriented language, e.g., Java. The name Python coined by Guido comes from the hit comedy show, *Monty Python's Flying Circus.*

Python is a general purpose programming language that supports multiple programming paradigms. The popular programming paradigms supported are as follows:

- Object oriented programming
- Functional programming
- Procedural programming
- Aspect oriented programming

A lot of object oriented programming concepts are present in Python, including classes, objects, data, and methods. Principles like abstraction, encapsulation, inheritance, and polymorphism can also be implemented and exhibited using Python. There are several advanced features in Python, including iterators, generators, list comprehensions, lambda expressions, and several modules like collections, itertools, and functools that provide the ability to write code following the functional programming paradigm. Python has been designed keeping in mind that simple and beautiful code is more elegant and easy to use than premature optimization and hard-to-interpret code.

Python's standard libraries are power-packed with a wide variety of capabilities and features, ranging from low-level hardware interfacing to handling files and working with text data. Easy extensibility and integration was considered when developing Python so that it can be easily integrated with existing applications and even rich application programming interfaces (APIs) can be created to provide interfaces to other applications and tools. Python also has a thriving and helpful developer community that ensures there are a ton of helpful resources and documentation out there on the Internet. The community also organizes various workshops and conferences throughout the world!

Python boosts productivity by reducing the time taken to develop, run, debug, deploy, and maintain large codebases compared to other languages like Java, C++, and C. Large programs of over a 100 lines can be reduced to 20 lines or less on average by porting them to Python. High-level abstractions help developers focus on the problem to be solved at hand rather than worry about language specific nuances. The hindrance of compiling and linking is also bypassed with Python. Hence, Python is often the best choice, especially when rapid prototyping and development is essential for solving an important problem in less time.

One of the main advantages of Python is that it is a multi-purpose programming language that can be used for just about anything! From web applications to intelligent systems, Python powers a wide variety of applications and systems. Besides being a multi-purpose language, the wide variety of frameworks, libraries, and platforms that have been developed using Python and used with Python form a complete robust ecosystem around it. These libraries make our lives easier by giving us a wide variety of capabilities and functionality to perform various tasks with minimal code. Some examples include libraries for handling databases, text data, machine learning, signal processing, image processing, deep learning, artificial intelligence, and the list goes on.

The Zen of Python

You might be wondering what on earth the Zen of Python is. However, if you are somewhat familiar with Python, this is one of the first things you'll get to know. The beauty of Python lies in its simplicity and elegance. "The Zen of Python" is a set of 20 guiding principles, also known as aphorisms, that have been influential behind Python's design. Long-time Pythoneer Tim Peters documented 19 of them in 1999 and they can be accessed at `https://hg.python.org/peps/file/tip/pep-0020.txt` as a part of the Python Enhancement Proposals (PEP) number 20 (PEP 20). The best part is, if you already have Python installed, you can access "The Zen of Python" by running the following code in the Python or IPython shell or a Jupyter notebook.

```
# zen of python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This output shows us the 19 principles that form the Zen of Python and is included in the Python language itself as an Easter egg. The principles are written in simple English and a lot of them are pretty self-explanatory even if you have not written code before. Many of them contain inside jokes! Python focuses on writing simple and clean code that's readable. It also encourages you to make sure you focus a lot on error handling and implementing code that's easy to interpret and understand. The one principle I would like you to remember is "simple is better than complex," which is applicable not only to Python but to a lot of things, especially when you are out there in the world solving problems. Sometimes a simple approach beats a more complex one as long as you know what you are doing.

Applications: When Should You Use Python?

Python—being a general and multi-purpose programming language—can be used to build applications and systems for different domains and solve diverse real-world problems. Python comes with a standard library that hosts a large number of useful libraries and modules that can be leveraged to solve various problems. Besides the standard library, there are thousands of third-party libraries that are readily available on the Internet encouraging open source and active development. The official repository that hosts third-party libraries and utilities for enhancing development in Python is the Python Package Index (PyPI) and you can access it by going to `https://pypi.python.org/` and checking out the various packages. Currently there are over 80,000 packages that you can install and use. While Python can be used for solving a lot of problems, we categorize some of the most popular domains and describe them as follows:

- **Scripting:** Python is often popularly known as a scripting language. It can be used to perform a wide variety of tasks like interfacing with networks and hardware, handling and processing files and databases, performing operating system (OS) operations, and receiving and sending e-mail.
- **Web development:** There are a lot of robust and stable Python frameworks out there that are used extensively for web development, such as Django, Flask, Web2Py, and Pyramid. You can use them to develop complete enterprise web applications and they support various architecture styles like RESTful APIs and the MVC architecture. They also provide ORM support to interact with databases and use object oriented programming on top of that. Python even has frameworks like Kivy, which support cross-platform development for developing apps on multiple platforms like iOS, Android, Windows, and OS X.
- **Graphical user interfaces:** A lot of desktop based applications with graphical user interfaces (GUIs) can be easily built with Python. Libraries and APIs like tkinter, PyQt, PyGTK, and wxPython allow developers to develop GUI-based apps with simple or complex interfaces. Various frameworks enable developers to create GUI-based apps for different OSes and platforms.

- **Systems programming:** Python, being a high-level language, has many interfaces to low-level OS services and protocols and the abstractions on top of these services enable developers to write robust and portable system monitoring and administration tools. You can use Python to perform several OS operations, including creating, handling, searching, deleting, and managing files and directories. The Python Standard Library (PSL) has OS and POSIX bindings, which can be used for handling files, multi-threading, multi-processing, environment variables, controlling sockets, pipes, and processes.
- **Database programming:** Python is used to connect and access data from different types of databases, be it SQL or NoSQL. APIs and connectors exist for these databases like MySQL, MSSQL, MongoDB, Oracle, PostgreSQL, and SQLite. In fact, SQLite, a lightweight relational database, now comes as a part of the Python standard distribution. Popular libraries like SQLAlchemy and SQLObject provide interfaces to access various relational databases and have ORM components to help implement OOP style classes and objects on top of relational tables.
- **Scientific computing:** Python really shows its flair for being multi-purpose in areas like numeric and scientific computing. You can perform simple as well as complex mathematical operations, including algebra and calculus. Libraries like SciPy and NumPy help researchers, scientists, and developers leverage highly optimized functions and interfaces for numeric and scientific programming. These libraries are also used as the base for developing complex algorithms in various domains like machine learning.
- **Machine learning and deep learning:** Python is regarded as one of the most popular languages today for machine learning. There exists a wide suite of libraries and frameworks like Scikit-Learn, h2o, TensorFlow, Keras, PyTorch, and even core libraries like NumPy and SciPy for not only implementing machine learning algorithms but also using them to solve real-world advanced analytics problems.

- **NLP and text analytics:** As mentioned, Python can handle text data really well and this has led to several popular libraries like NLTK, Gensim, and spaCy for natural language processing, information retrieval, and text analytics. You can also apply standard machine learning algorithms to solve problems related to text analytics. We explore several of these libraries in this book.

Even though this list might seem a bit overwhelming, this is just scratching the surface of what is possible with Python. It is widely used in several other domains including artificial intelligence, game development, robotics, internet of things, computer vision, media processing, and network and system monitoring, just to name a few. To know some of the widespread success stories achieved with Python in different diverse domains like arts, science, computer science, education and others, enthusiastic programmers and researchers can check out this link `https://www.python.org/about/success/`. If you want to know about various popular applications developed using Python, you can check out `https://www.python.org/about/apps/` and `https://wiki.python.org/moin/Applications`, where you will definitely find some applications that you have used.

Drawbacks: When Should You Not Use Python?

Just like any tool or language out there, Python has its own advantages and disadvantages, and in this section we highlight some of them so that you are aware of them when developing and writing code in Python.

- **Execution speed performance:** Performance is a pretty heavy term and can mean several things, so we pinpoint the exact area we want to talk about and that is execution speed. Since Python is not a fully compiled language, it will always be slower than low-level fully compiled programming languages like C and C++. There are several ways you can optimize your code, including multi-threading and multi-processing and using static typing and C extensions for Python, also known as Cython. You can also consider using PyPy, which is much faster than normal Python since it uses a Just in Time (JIT) compiler. See `http://pypy.org/`. Often, if you write well optimized code, you can develop applications in Python just fine that do not need to depend on other languages. Remember that often the problem is not with the tool but with the code you write, something all developers and engineers realize with time and experience.

- **Global interpreter lock:** The Global Interpreter Lock (GIL) is a mutual exclusion lock used is several programming language interpreters like Python and Ruby. Interpreters using GIL only allow one single thread to effectively execute at a time even when run on a multi-core processor. This limits the effectiveness of parallelism achieved by multi-threading depending on whether the processes are I/O bound or CPU bound and how many calls it makes outside the interpreter. Python 3.x also has modules like asyncio, which can be used for asynchronous IO operations.
- **Version incompatibility:** If you have been following Python closely, you will know that once Python released the 3.x version from 2.7.x, it was backward incompatible in several aspects and it opened a huge can of worms. Several major libraries and packages had been built in Python 2.7.x and they started breaking when users unknowingly updated their Python versions. Code deprecation and version changes are some of the most important factors in systems breaking down. However, Python 3.x is quite stable now and the majority of the userbase has started using it actively in the last couple of years.

Many of these issues are not specific to Python but to other languages too. Hence, you should not be discouraged from using Python just because of these points. That said, you should definitely remember them when writing code and building systems.

Python Implementations and Versions

There are several implementations of Python and different versions of Python that are released periodically since it is under active development. We discuss implementations and versions and their significance, and this should give you some idea of what Python environments exist and which ones you might want to use for your development needs. Currently, there are four major production-ready, robust, and stable implementations of Python:

- **CPython:** This is the regular old Python we know as just Python. It is both a compiler and interpreter and comes with its own set of standard packages and modules that have all been written in standard C. This version can be used directly in all popular modern platforms. Most of the Python third-party packages and libraries are compatible with this version.
- **PyPy:** A faster alternative Python implementation that uses a Just-in-Time (JIT) compiler to make code run faster than the CPython implementation, sometimes giving speedups in the range of 10x - 100x. It is also more memory efficient, supporting greenlets and stackless for high parallelism and concurrency.
- **Jython:** A Python implementation for the Java platform, supporting Java Virtual Machine (JVM) for any version of Java ideally above version 7. Using Jython you can write code leveraging all types of Java libraries, packages, and frameworks. It works best when you know more about the Java syntax and the OOP principles that are used extensively in Java like classes, objects, and interfaces.
- **IronPython:** The Python implementation for the popular Microsoft .NET framework, also called the Common Language Runtime (CLR). You can use all of Microsoft's CLR libraries and frameworks in IronPython and even though you do not essentially have to write code in C#, it is useful to know more about syntax and constructs for C# to use IronPython effectively.

To start, I suggest you use the default Python, which is the CPython implementation and experiment with the other versions only if you are interested in interfacing with other languages like C# and Java and need to use them in your codebase.

There are two major Python versions—the 2.x series and the 3.x series, where x is a number. Python 2.7 was the last major version in the 2.x series released in 2010 and from then on, future releases have bug fixes and performance improvements but no new features. A very important point to remember is that support for Python 2.x is ending by 2020. The 3.x series started with Python 3.0, which introduced many backward incompatible changes compared to Python 2.x and each version 3 release not only has bug fixes and improvements but also introduces new features like the AsyncIO module. At the time of writing this, Python 3.7 was the latest version in the 3.x series and it was released in June 2018. There are many arguments over which version of Python should be used. Considering that support for Python 2 is ending by 2020 and no new features or enhancements are planned for it, we recommend you use Python 3 for all your projects, research, and development.

Setting Up a Robust Python Environment

Now that you have been acquainted with Python and know more about the language, its capabilities, implementations, and versions, this section covers some essentials on how to set up your development environment and handle package management and virtual environments. This section gives you a good head start on getting things ready for following along with the various hands-on examples covered in this book.

Which Python Version?

We have previously talked about two major Python versions—the 2.x series and the 3.x series. Both the versions are quite similar; however, there have been several backward incompatible changes in the 3.x version, which has led to a huge drift between people who use 2.x and people who use 3.x. Most legacy code and a large majority of Python packages on PyPI are developed in Python 2.7.x and the package owners do not have the time or the will to port all their codebases to Python 3.x. Some of the changes in 3.x are as follows:

- All text strings are Unicode by default
- `print` and `exec` are now functions and no longer statements
- Several methods like `range()` return a memory-efficient iterable instead of a list
- The style for classes have changed
- Libraries and names have changed based on convention and style violations
- More features, modules, and enhancements

To learn more about all the changes introduced in Python 3.x, check out https://docs.python.org/3/whatsnew/3.7.html, which is the official documentation listing the changes. This should give you a pretty good idea of what changes can break your code if you are porting it from Python 2 to Python 3.

Now addressing the problem of selecting which version, we definitely recommend using Python 3 at all times. The primary reason behind this is a recent announcement from the Python core group members that mentioned that support for Python 2 will be ending by 2020 and no new features or enhancements will be pushed to Python 2. We recommend checking out PEP 373 `https://legacy.python.org/dev/peps/pep-0373/`, which covers this issue in further detail. However, if you are working on a large legacy codebase with Python 2.x, you might need to stick to it until porting it to Python 3 is possible. We use Python 3.x in this book and recommend you do the same. For code in Python 2.x, you can refer to the previous release of this book as needed.

Which Operating System?

There are several popular operating systems (OSes) out there and each person has their own preference. The beauty of Python is that is can run seamlessly on any OS without much hassle. Some of the different OSes that Python supports include:

- Windows
- Linux
- MacOS (also known as OS X)

You can choose any OS of your choice and use it to following along with the examples. We use a combination of Linux and Windows as our OS platforms. Python external packages are typically easy to install on UNIX-based OSes like Linux and MacOS. However, sometimes there are major issues in installing them on Windows, so we highlight such instances and address them so that executing any of the code snippets and samples here becomes easy for our Windows readers. You are most welcome to use any OS of your choice when following the examples in this book!

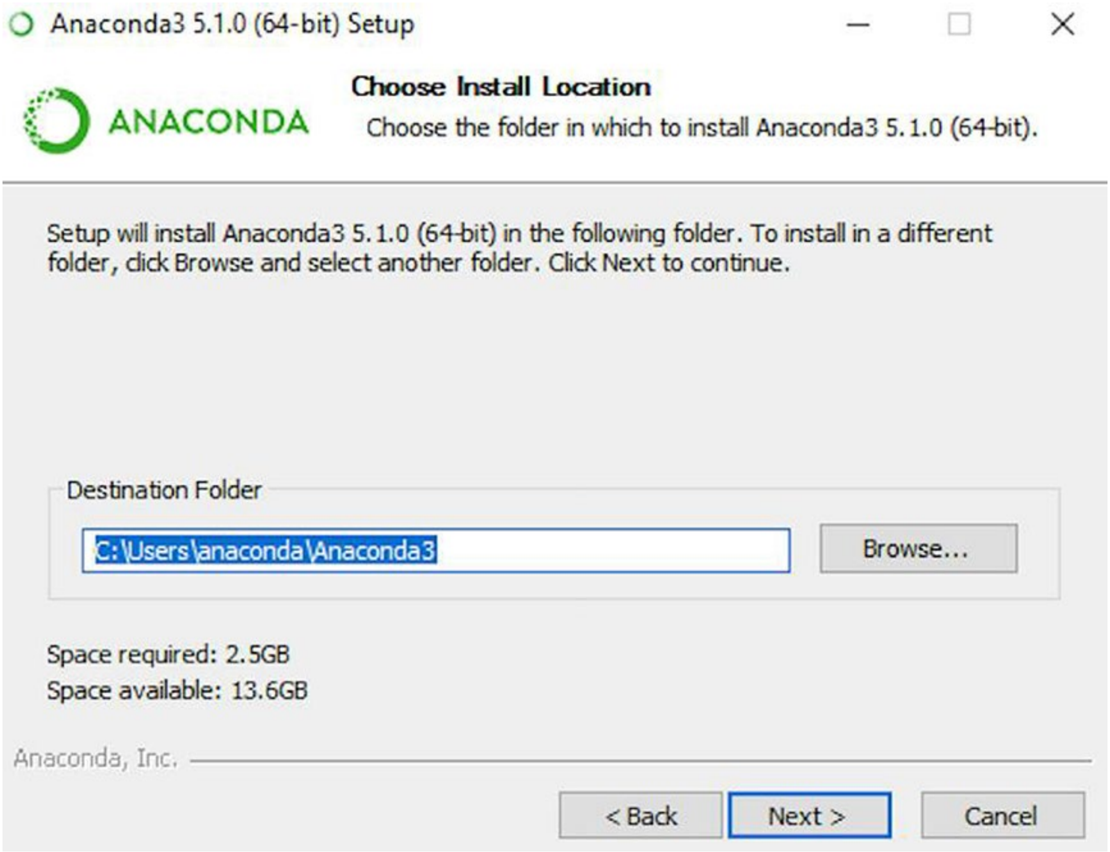
Integrated Development Environments

Integrated development environments (IDEs) are software products that enable developers to be highly productive by providing a complete suite of tools and capabilities necessary for writing, managing, and executing code. The usual components of an IDE include a source editor, debugger, compiler, interpreter, and refactoring and build tools. They also have other capabilities like code-completion, syntax highlighting, error highlighting and checks, objects, and variable explorers. IDEs can be used to manage entire codebases and are much better than trying to write code in a simple text editor, which takes more time. However, more experienced developers often use simple plain text editors to write code, especially if they are working in server environments. The https://wiki.python.org/moin/IntegratedDevelopmentEnvironments link provides a list of IDEs used specially with Python. We use a combination of PyCharm, Spyder, Sublime Text and Jupyter notebooks. The code examples in this book are demonstrated on Jupyter notebooks mostly, which comes installed along with the Anaconda Python distribution for writing and executing the code.

Environment Setup

In this section, we cover details regarding how to set up your Python environment with minimal effort and the main required components. You can head over to the official Python website and download Python 3.7 from https://www.python.org/downloads/ or you can download a complete Python distribution with over hundreds of packages specially built for data science and AI, known as the Anaconda Python distribution, from Anaconda (formerly known as Continuum Analytics). This provides a lot of advantages, especially to Windows users, where installing some of the packages like NumPy and SciPy can sometimes cause major issues. You can get more information about Anaconda and the excellent work they are doing by visiting https://www.anaconda.com. Anaconda comes with conda, an open source package and environment management system and Spyder (Scientific Python Development Environment), an IDE for writing and executing your code.

To start your environment setup, you can follow along the instructions mentioned at https://docs.anaconda.com/anaconda/install/windows for Windows or use the instructions for any other OS of your choice. Head over to https://www.anaconda.com/download/ and download the 64- or 32-bit Python 3 installer for Windows, depending on your OS version. For other OSes, you can check the relevant instructions on the website. Start the executable and follow the instructions on the screen by clicking the Next button at each stage. Remember to set a proper install location, as depicted in Figure 2-1.


Anaconda3 5.1.0 (64-bit) Setup
ANACONDA
Choose Install Location
Choose the folder in which to install Anaconda3 5.1.0 (64-bit).

Setup will install Anaconda3 5.1.0 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

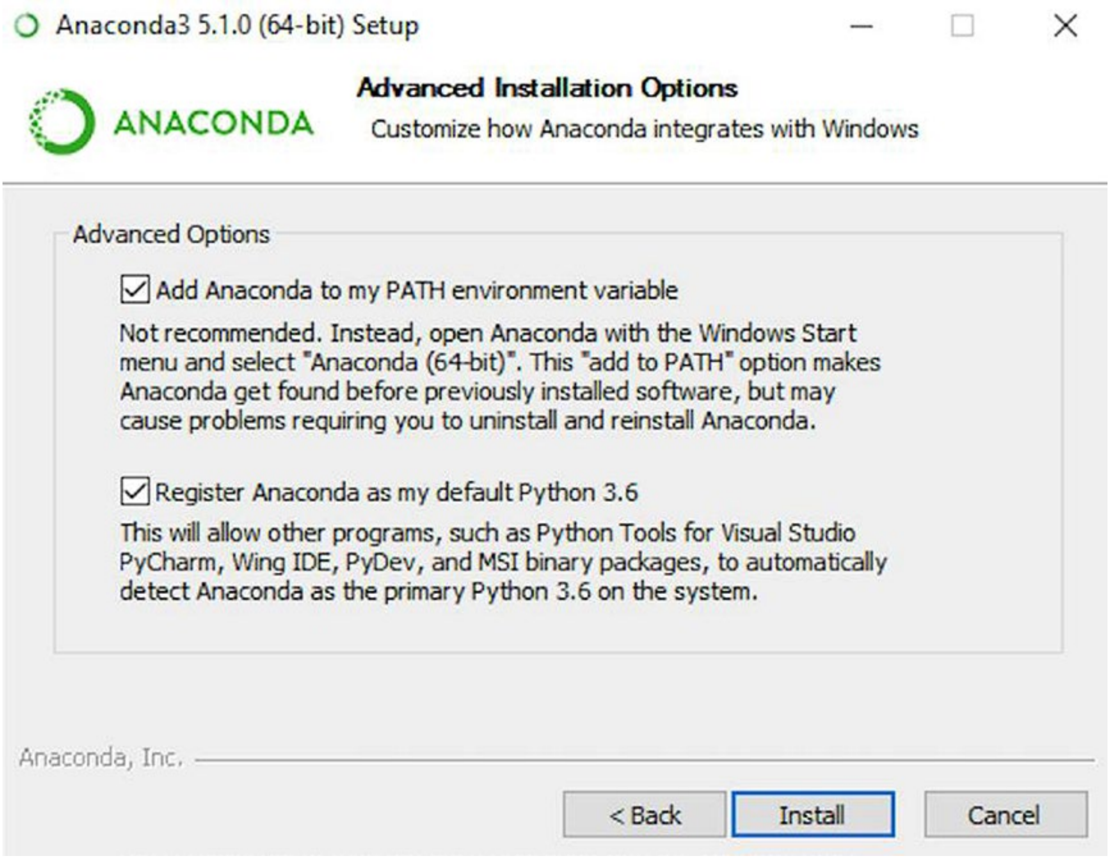
Destination Folder: C:\Users\anaconda\Anaconda3

Space required: 2.5GB
Space available: 13.6GB

Browse...
Anaconda, Inc.
< Back
Next >
Cancel

Figure 2-1. *Installing the Anaconda Python distribution: setting up an install location*

Before starting the actual installation, remember to check the two options shown in Figure 2-2.



Anaconda3 5.1.0 (64-bit) Setup
ANACONDA
Advanced Installation Options
Customize how Anaconda integrates with Windows
Anaconda, Inc.
< Back
Install
Cancel

Advanced Options

☑ Add Anaconda to my PATH environment variable

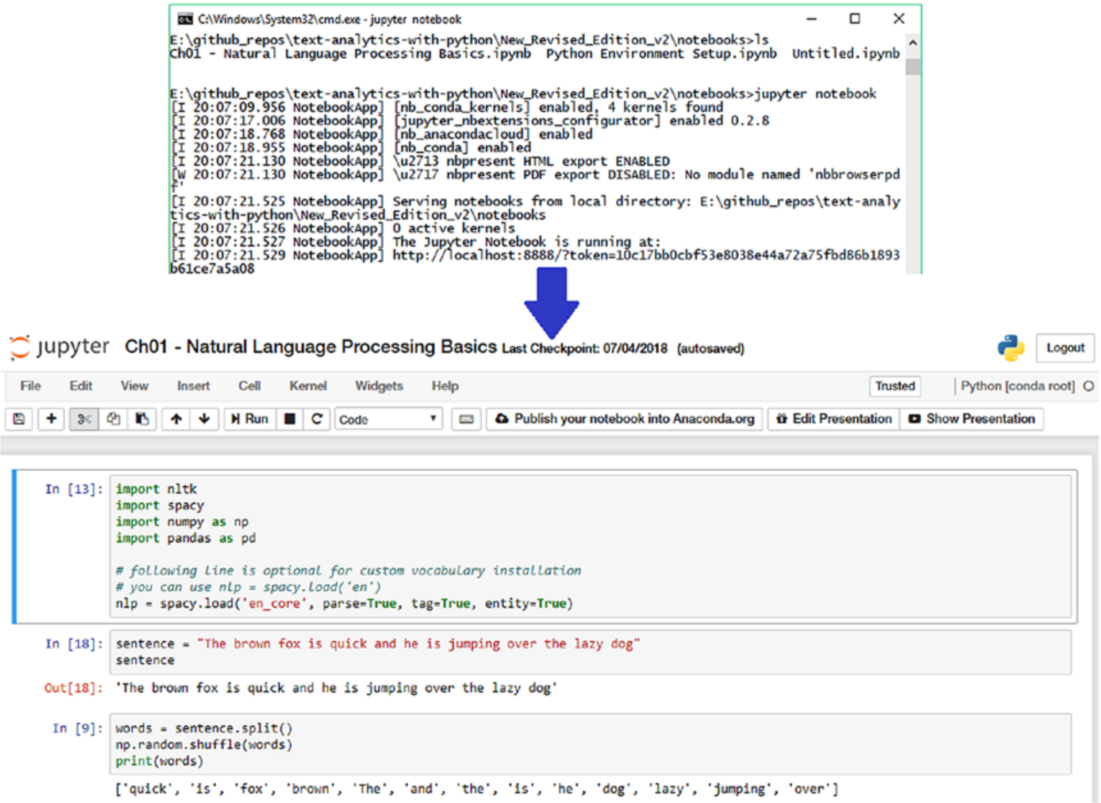
Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

☑ Register Anaconda as my default Python 3.6

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.6 on the system.

Figure 2-2. *Installing the Anaconda Python distribution: adding to system path*

Once the installation is complete, you can either start Spyder by double-clicking the relevant icon or start the Python or IPython shell from the command prompt. To start Jupyter notebooks, you can just type `jupyter notebook` from a terminal or command prompt. Spyder provides you with a complete IDE to write and execute code in both the regular Python and the IPython shell. Figure 2-3 shows you how to start a Jupyter notebook.

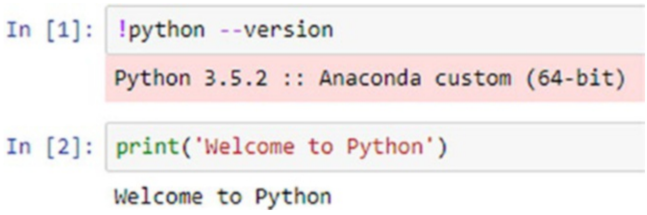

C:\Windows\System32\cmd.exe - jupyter notebook
jupyter
Ch01 - Natural Language Processing Basics Last Checkpoint: 07/04/2018 (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python [conda root]
Run
Code
Publish your notebook into Anaconda.org
Edit Presentation
Show Presentation

```
E:\github_repos\text-analytics-with-python\New_Revised_Edition_v2\notebooks>ls
Ch01 - Natural Language Processing Basics.ipynb  Python Environment Setup.ipynb  Untitled.ipynb

E:\github_repos\text-analytics-with-python\New_Revised_Edition_v2\notebooks>jupyter notebook
[I 20:07:09.956 NotebookApp] [nb_conda_kernels] enabled, 4 kernels found
[I 20:07:17.006 NotebookApp] [jupyter_nbextensions_configurator] enabled 0.2.8
[I 20:07:18.768 NotebookApp] [nb_anacondacloud] enabled
[I 20:07:18.955 NotebookApp] [nb_conda] enabled
[I 20:07:21.130 NotebookApp] \u2713 nbpresent HTML export ENABLED
[W 20:07:21.130 NotebookApp] \u2717 nbpresent PDF export DISABLED: No module named 'nbbrowserpd
f'
[I 20:07:21.525 NotebookApp] Serving notebooks from local directory: E:\github_repos\text-analy
tics-with-python\New_Revised_Edition_v2\notebooks
[I 20:07:21.526 NotebookApp] 0 active kernels
[I 20:07:21.527 NotebookApp] The Jupyter Notebook is running at:
[I 20:07:21.529 NotebookApp] http://localhost:8888/?token=10c17bb0cbf53e8038e44a72a75fbd86b1893
b61ce7a5a08
```

```
In [13]: import nltk
         import spacy
         import numpy as np
         import pandas as pd

         # following line is optional for custom vocabulary installation
         # you can use nlp = spacy.load('en')
         nlp = spacy.load('en_core', parse=True, tag=True, entity=True)

In [18]: sentence = "The brown fox is quick and he is jumping over the lazy dog"
         sentence

Out[18]: 'The brown fox is quick and he is jumping over the lazy dog'

 In [9]: words = sentence.split()
         np.random.shuffle(words)
         print(words)

         ['quick', 'is', 'fox', 'brown', 'The', 'and', 'the', 'is', 'he', 'dog', 'lazy', 'jumping', 'over']
```

Figure 2-3. *Starting a Jupyter notebook*

This should give you an idea how easy it is to showcase code and outputs interactively in Jupyter notebooks. We showcase our code usually through these notebooks. To test if Python is properly installed, you can simply type `python --version` from the terminal or even from your notebook, as depicted in Figure 2-4.

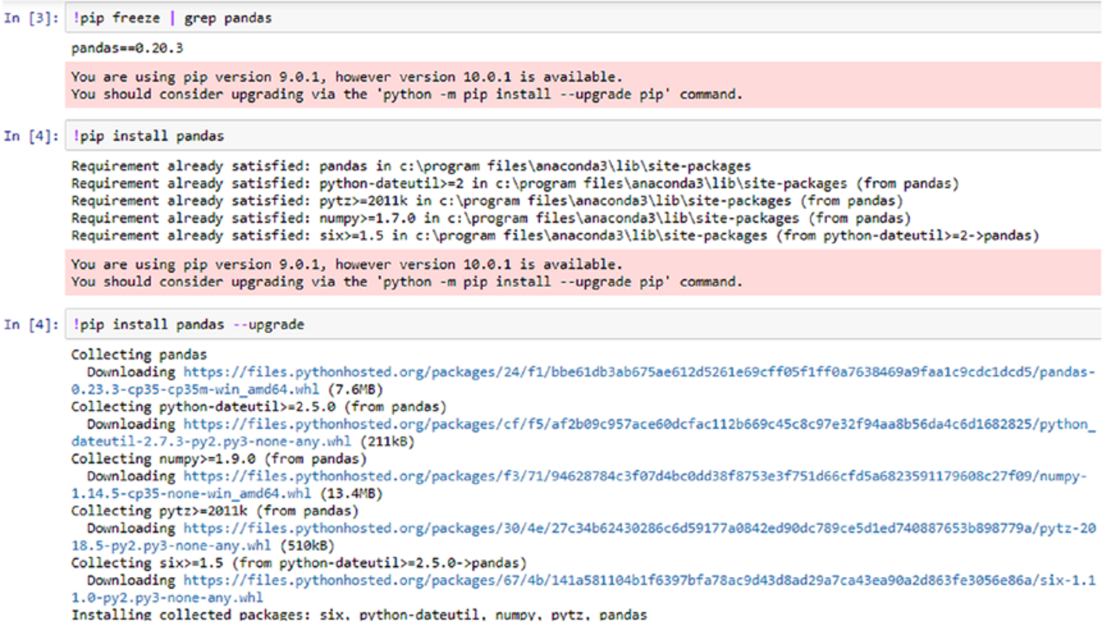


```
In [1]: !python --version

        Python 3.5.2 :: Anaconda custom (64-bit)

In [2]: print('Welcome to Python')

        Welcome to Python
```

Figure 2-4. *Checking your Python installation*

Package Management

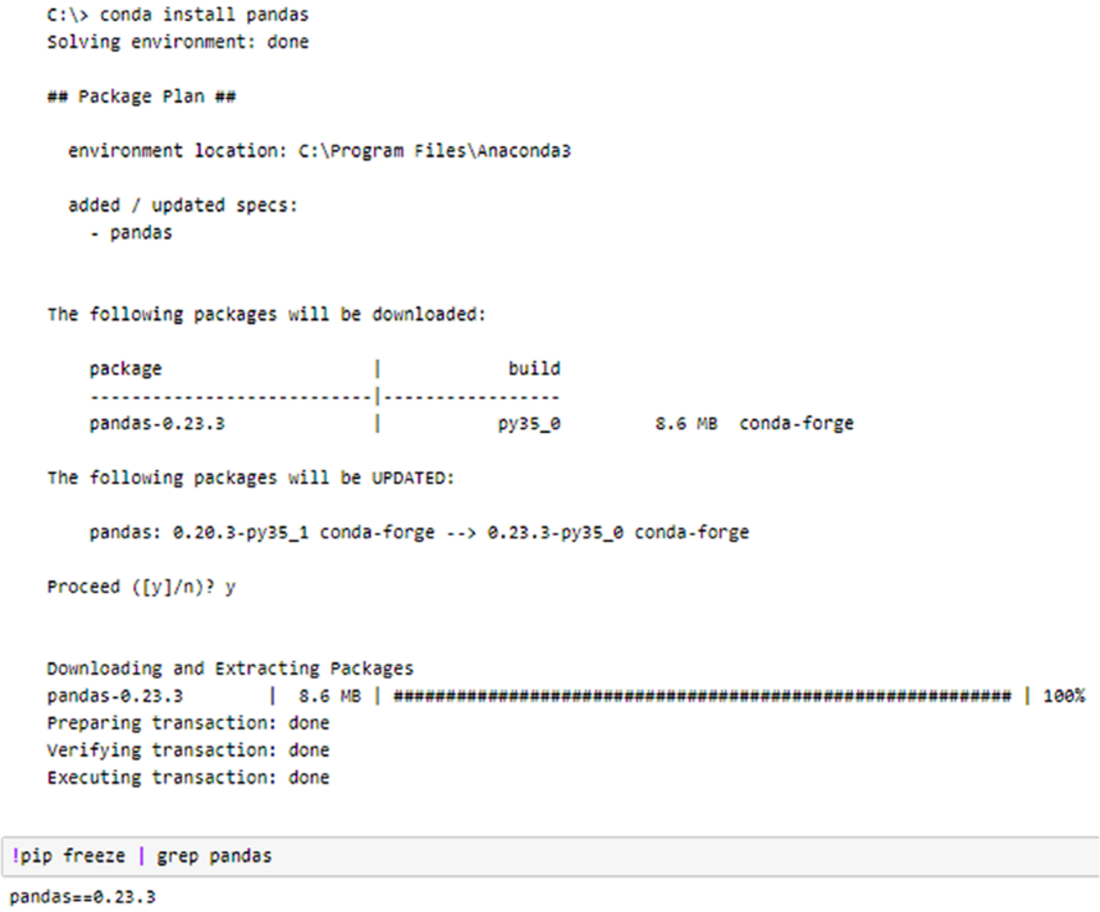
We now cover package management briefly. You can use the `pip` or `conda` command to install, uninstall, and upgrade packages. The following shell command depicts installing the pandas library via `pip`. You can check if a package is installed using the `pip freeze <package_name>` command and install packages using the `pip install <package_name>` command, as depicted in Figure 2-5. If you already have a package/library installed, you can use the `--upgrade` flag.



```
In [3]: !pip freeze | grep pandas

        pandas==0.20.3

        You are using pip version 9.0.1, however version 10.0.1 is available.
        You should consider upgrading via the 'python -m pip install --upgrade pip' command.

In [4]: !pip install pandas

        Requirement already satisfied: pandas in c:\program files\anaconda3\lib\site-packages
        Requirement already satisfied: python-dateutil>=2 in c:\program files\anaconda3\lib\site-packages (from pandas)
        Requirement already satisfied: pytz>=2011k in c:\program files\anaconda3\lib\site-packages (from pandas)
        Requirement already satisfied: numpy>=1.7.0 in c:\program files\anaconda3\lib\site-packages (from pandas)
        Requirement already satisfied: six>=1.5 in c:\program files\anaconda3\lib\site-packages (from python-dateutil>=2->pandas)

        You are using pip version 9.0.1, however version 10.0.1 is available.
        You should consider upgrading via the 'python -m pip install --upgrade pip' command.

In [4]: !pip install pandas --upgrade

        Collecting pandas
          Downloading https://files.pythonhosted.org/packages/24/f1/bbe61db3ab675ae612d5261e69cff05f1ff0a7638469a9faa1c9cdc1dcd5/pandas-
        0.23.3-cp35-cp35m-win_amd64.whl (7.6MB)
        Collecting python-dateutil>=2.5.0 (from pandas)
          Downloading https://files.pythonhosted.org/packages/cf/f5/af2b09c957ace60dcfac112b669c45c8c97e32f94aa8b56da4c6d1682825/python_
        dateutil-2.7.3-py2.py3-none-any.whl (211kB)
        Collecting numpy>=1.9.0 (from pandas)
          Downloading https://files.pythonhosted.org/packages/f3/71/94628784c3f07d4bc0dd38f8753e3f751d66cfd5a6823591179608c27f09/numpy-
        1.14.5-cp35-none-win_amd64.whl (13.4MB)
        Collecting pytz>=2011k (from pandas)
          Downloading https://files.pythonhosted.org/packages/30/4e/27c34b62430286c6d59177a0842ed90dc789ce5d1ed740887653b898779a/pytz-20
        18.5-py2.py3-none-any.whl (510kB)
        Collecting six>=1.5 (from python-dateutil>=2.5.0->pandas)
          Downloading https://files.pythonhosted.org/packages/67/4b/141a581104b1f6397bfa78ac9d43d8ad29a7ca43ea90a2d863fe3056e86a/six-1.1
        1.0-py2.py3-none-any.whl
        Installing collected packages: six, python-dateutil, numpy, pytz, pandas
```

Figure 2-5. *Python package management with pip*

The conda package manager is better than pip in several aspects, since it provides a holistic view of what dependencies are going to be upgraded and the specific versions and other details during installation. pip often fails to install some packages in Windows; however, conda usually has no such issues during installation. Figure 2-6 depicts how to install and manage packages using conda.



```
C:\> conda install pandas
Solving environment: done

## Package Plan ##

  environment location: C:\Program Files\Anaconda3

  added / updated specs:
    - pandas


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    pandas-0.23.3              |           py35_0         8.6 MB  conda-forge

The following packages will be UPDATED:

    pandas: 0.20.3-py35_1 conda-forge --> 0.23.3-py35_0 conda-forge

Proceed ([y]/n)? y


Downloading and Extracting Packages
pandas-0.23.3        |  8.6 MB | ########################################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

```
!pip freeze | grep pandas
```

```
pandas==0.23.3
```

Figure 2-6. *Python package management with conda*

Now you have a much better idea of how to install external packages and libraries in Python. This will be useful later, whenever you want to install external Python packages in your environment. Your Python environment should now be set up and ready for executing code. Before we dive into techniques for handling text data in Python, we conclude with a discussion about virtual environments.

Virtual Environments

A virtual environment, also called a *venv,* is a complete isolated Python environment with its own Python interpreter, libraries, modules, and scripts. This environment is a standalone environment isolated from other virtual environments and the default system level Python environment. Virtual environments are extremely useful when you have multiple projects or codebases that have dependencies on different versions of the same packages or libraries. For example, if my project called `TextApp1` depends on NLTK 2.0 and another project called `TextApp2` depends on NLTK 3.0, then it would be impossible to run both projects on the same system. Hence the need for virtual environments that provide complete isolated environments, which can be activated and deactivated as needed.

To set up a virtual environment, you need the `virtualenv` package. We create a new directory where we want to keep our virtual environment and install `virtualenv` as follows:

```
E:\>mkdir Apress
E:\>cd Apress
E:\Apress>pip install virtualenv

Collecting virtualenv
Installing collected packages: virtualenv
Successfully installed virtualenv-16.0.0
```

Once the package is installed, you can create a virtual environment. Here we create a new project directory called `test_proj` and create the virtual environment inside the directory:

```
E:\Apress>mkdir test_proj && chdir test_proj
E:\Apress\test_proj>virtualenv venv

Using base prefix 'c:\\program files\\anaconda3'
New python executable in E:\Apress\test_proj\venv\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

Now that you have installed the virtual environment successfully, let's try to observe the major differences between the global system Python environment and our virtual environment. If you remember, we updated our global system Python's pandas package version to 0.23 in the previous section. We can verify it using the following commands.

```
E:\Apress\test_proj>echo 'This is Global System Python'
'This is Global System Python'

E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.23.3
```

Now supposed we wanted an older version of pandas in our virtual environment but we don't want to affect our global system Python environment. We can do this by activating our virtual environment and installing pandas.

```
E:\Apress\test_proj>venv\Scripts\activate
(venv) E:\Apress\test_proj>echo 'This is VirtualEnv Python'
'This is VirtualEnv Python'

(venv) E:\Apress\test_proj>pip install pandas==0.21.0
Collecting pandas==0.21.0
    100% |################################| 9.0MB 310kB/s
Collecting pytz>=2011k (from pandas==0.21.0)
Collecting python-dateutil>=2 (from pandas==0.21.0)
Collecting numpy>=1.9.0 (from pandas==0.21.0)
Collecting six>=1.5 (from python-dateutil>=2->pandas==0.21.0)

Installing collected packages: pytz, six, python-dateutil, numpy, pandas
Successfully installed numpy-1.14.5 pandas-0.21.0 python-dateutil-2.7.3
pytz-2018.5 six-1.11.0

(venv) E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.21.0
```

For other OS platforms, you might need to use the command `source venv/bin/activate` to activate the virtual environment. Once the virtual environment is active, you can see the `(venv)` notation, as shown in the preceding code output, and any new packages you install will be placed in the `venv` folder in complete isolation from the global system Python environment.

You can see from the previous code snippets how the pandas package has different versions in the same machine—0.23.3 for global Python and 0.21.0 for the virtual environment Python. Hence, these isolated virtual environments can run seamlessly on the same system. Once you have finished working in the virtual environment, you can deactivate it again as follows.

```
(venv) E:\Apress\test_proj>venv\Scripts\deactivate
E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.23.3
```

This will bring you back to the system's default Python version with all its installed libraries and, as expected, the pandas version is the newer one that we had installed. This gives us a good idea about the utility and advantages of virtual environments and once you start working on several projects, you should definitely consider using it. To learn more about virtual environments, check out `http://docs.python-guide.org/en/latest/dev/virtualenvs/`, which is the official documentation for the `virtualenv` package. This brings us to the end of our installation and setup activities. Next, we look into some basic concepts around Python syntax and structure before diving into handling text data with Python using hands-on examples.

Python Syntax and Structure

We discuss briefly the basic syntax, structure, and design philosophies that are followed when writing Python code for applications and systems. There is a defined hierarchical syntax for Python code, which you should remember when writing code. Any big Python application or system is built using several modules, which are themselves comprised of Python statements. Each statement is like a command or direction to the system directing what operations it should perform. These statements are comprised of expressions and objects. Everything in Python is an object, including functions, data structures, types, classes, and so on. This hierarchy can be visualized better in Figure 2-7.

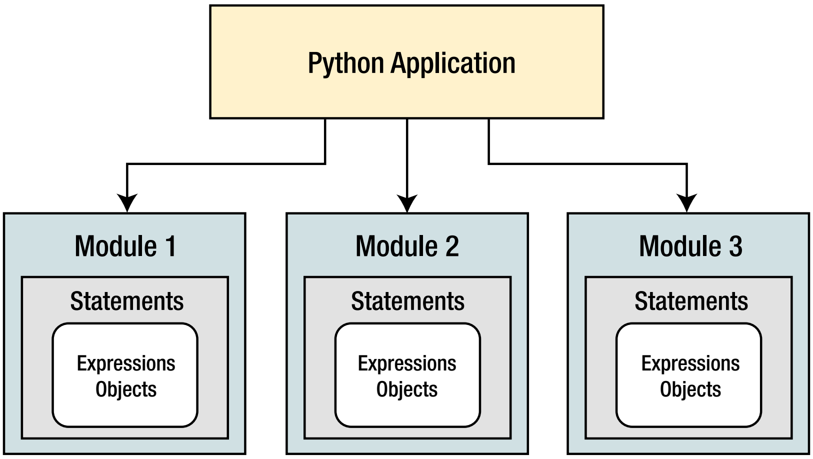


Figure 2-7. *Python program structure hierarchy*

The basic statements consist of objects and expressions (which use objects and process and perform operations on them). Objects can be anything from simple data types and structures to complex objects, including functions and reserved words that have their own specific roles. Python has around 30+ keywords or reserved words, all of which have their own designated role and function. We assume that you have some knowledge of basic programming constructs, but in case you do not, don't despair! In the next section, we showcase how to work with text data using detailed hands-on examples.

Working with Text Data

In this section, we briefly cover specific data types tailored to handle text data and how these data types and their associated utilities, functions, and methods will be useful in the subsequent chapters. The main data types that are used to handle text data in Python are strings. These can be normal strings, bytes storing binary information, or Unicode. By default, all strings are Unicode in Python 3.x but are not so in Python 2.x. This is something you should definitely keep in mind when dealing with text in different Python distributions.

Strings are a sequence of characters in Python, similar to arrays and code with a set of attributes and methods that can be leveraged to manipulate and operate on text data easily. This makes Python the language of choice for text analytics in many scenarios. There are various types of strings that we discuss, with several examples in the next section.

String Literals

There are various types of strings, as mentioned earlier. The following BNF (Backus-Naur Form) gives us the general lexical definitions for producing strings, as seen in the official Python docs.

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
                     | "b" | "B" | "br" | "Br" | "bR" | "BR"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem*
                     '"""'
```

```
shortstringitem ::=  shortstringchar | escapeseq
longstringitem  ::=  longstringchar | escapeseq
shortstringchar ::=  <any source character except "\" or newline or the
                     quote>
longstringchar  ::=  <any source character except "\">
escapeseq       ::=  "\" <any ASCII character>
```

These rules tell us that different types of string prefixes exist and can be used with different string types to produce string literals. In simple terms, the following types of string literals are used the most:

- **Short strings:** These strings are usually enclosed with single quotes (`'`) or double quotes (`"`) around the characters. Some examples are `'Hello'` and `"Hello"`.
- **Long strings:** These strings are usually enclosed with three single (`'''`) or double quotes (`"""`) around the characters. Some examples are `"""Hello, I'm a long string"""` or `'''Hello I\'m a long string '''`. Note the (`\'`) indicates an escape sequence which we shall talk about soon.
- **Escape sequences in strings:** These strings often have escape sequences embedded in them, where the rule for escape sequences starts with a backslash (`\`) followed by any ASCII character. Hence, they perform backspace interpolation. Popular escape sequences include (`\n`), which indicates a newline character and (`\t`), indicating a tab.
- **Bytes:** These are used to represent bytestrings that create objects of the byte's data type. These strings can be created as `bytes('...')` or using the `b'...'` notation. Examples include `bytes('hello')` and `b'hello'`.
- **Raw strings:** These strings were originally created specifically for regular expressions (regex) and regex patterns. These strings can be created using the `r'...'` notation and keep the string in its raw or native form. Hence, they do not perform any backspace interpolation and turn off the escape sequences. An example is `r'Hello'`.

- **Unicode:** These strings support Unicode characters in text and they are usually non-ASCII character sequences. These strings are denoted with the `u'...'` notation. However, in Python 3.x all string literals are typically represented as Unicode. Besides the string notation, there are several specific ways to represent special Unicode characters in the string. The usual include the hex byte value escape sequence of the format `'\xVV'`. Besides this, we also have Unicode escape sequences of the form `'\uVVVV'` and `'\uVVVVVVVV'`, where the first form uses four hex-digits for encoding a 16 bit character and the second uses eight hex-digits for encoding a 32-bit character. Some examples include `u 'H\xe8llo'` and `u 'H\u00e8llo'`, which represents the string `'Hèllo'`.

Now that we know the main types of string literals, let's look at ways to represent strings.

Representing Strings

Strings are sequences or collections of characters that are used to store and represent textual data, which is our data type of choice in most of the book's examples. Strings can be used to store text and bytes as information. Strings have a wide variety of methods that can be used for handling and manipulating strings, which we see in a subsequent section. An important point to remember is that strings are immutable and any operations performed on strings create a new string object (which can be checked using the `id` function) rather than just changing the value of the existing string object. Let's look at some basic string representations.

```
new_string = "This is a String"  # storing a string
print('ID:', id(new_string))  # shows the object identifier (address)
print('Type:', type(new_string))  # shows the object type
print('Value:', new_string)  # shows the object value
```

```
ID: 1907471142032
Type: <class 'str'>
Value: This is a String
```

```
# simple string
simple_string = 'Hello!' + " I'm a simple string"
print(simple_string)
```

```
Hello! I'm a simple string
```

Representing multi-line strings is also quite easy and can be done as follows:

```
# multi-line string, note the \n (newline) escape character automatically
created
multi_line_string = """Hello I'm
a multi-line
string!"""
```

```
multi_line_string
```

```
"Hello I'm\na multi-line\nstring!"
```

```
print(multi_line_string)
```

```
Hello I'm
a multi-line
string!
```

Let's now look at ways to represent escape sequences and raw strings in their native format, without any escape sequences.

```
# Normal string with escape sequences leading to a wrong file path!
escaped_string = "C:\the_folder\new_dir\file.txt"
print(escaped_string)  # will cause errors if we try to open a file here
```

```
C: he_folder
ew_dirile.txt
```

```
# raw string keeping the backslashes in its normal form
raw_string = r'C:\the_folder\new_dir\file.txt'
print(raw_string)
```

```
C:\the_folder\new_dir\file.txt
```

Let’s look at ways to represent non-ASCII characters leveraging Unicode, which can also be used to represent symbols like emojis.

```
# unicode string literals
string_with_unicode = 'H\u00e8llo!'
print(string_with_unicode)
```

```
Hèllo!
```

```
more_unicode = 'I love Pizza 🍕!  Shall we book a cab 🚕 to get pizza?'
print(more_unicode)
```

```
I love Pizza 🍕!  Shall we book a cab 🚕 to get pizza?
```

```
print(string_with_unicode + '\n' + more_unicode)
```

```
Hèllo!
I love Pizza 🍕!  Shall we book a cab 🚕 to get pizza?
```

```
' '.join([string_with_unicode, more_unicode])
```

```
'Hèllo! I love Pizza 🍕!  Shall we book a cab 🚕 to get pizza?'
```

```
more_unicode[::-1]  # reverses the string
```

```
'?azzip teg ot 🚕 bac a koob ew llahS  !🍕 azziP evol I'
```

String Operations and Methods

Strings are iterable sequences and hence a lot of operations can be performed on them. This is especially helpful when processing and parsing textual data into easy-to-consume formats. There are several operations that can be performed on strings. We have categorized them into the following segments.

- Basic operations
- Indexing and slicing
- Methods
- Formatting
- Regular expressions

These cover the most frequently used techniques for working with strings and form the base of what we would need to get started. In the next chapter, we look at understanding and processing textual data based on concepts we learned in the first two chapters.

Basic Operations

There are several basic operations you can perform on strings, including concatenation and checking for substrings, characters, and lengths. We start off with some basic examples of concatenating strings.

```
In [10]: 'Hello 😊' + ' and welcome ' + 'to Python 🐍!'
Out [10]: 'Hello 😊 and welcome to Python 🐍!'

In [11]: 'Hello 😊' ' and welcome ' 'to Python 🐍!'
Out [11]: 'Hello 😊 and welcome to Python 🐍!'
```

Let's now look at some ways of concatenating variables and literals when handling strings.

```
# concatenation of variables and literals
In [12]: s1 = 'Python 💻!'
    ...: 'Hello 😊 ' + s1
Out [12]: 'Hello 😊 Python 💻!'

In [13]: 'Hello 😊 ' s1
File "<ipython-input-17-da1762b9f01f>", line 1
    'Hello 😊 ' s1
                ^
SyntaxError: invalid syntax
```

We now look at some more ways of concatenating strings.

```
In [5]: s2 = '--🐍Python🐍--'
   ...: s2 * 5
Out [5]: '--🐍Python🐍----🐍Python🐍----🐍Python🐍----🐍Python🐍----
🐍Python🐍--'

In [6]: s1 + s2
Out [6]: 'Python 💻!--🐍Python🐍--'

In [7]: (s1 + s2)*3
Out [7]: 'Python 💻!--🐍Python🐍--Python 💻!--🐍Python🐍--Python 💻
!--🐍Python🐍--'
```

```
# concatenating several strings together in parentheses
In [8]: s3 = ('This '
   ...:       'is another way '
   ...:       'to concatenate '
   ...:       'several strings!')
   ...: s3
Out[8]: 'This is another way to concatenate several strings!'
```

Here are some more essential operations like checking for substrings and finding the length of a typical string.

```
In [9]: 'way' in s3
Out[9]: True

In [10]: 'python' in s3
Out[10]: False

In [11]: len(s3)
Out[11]: 51
```

Indexing and Slicing

We have already discussed that strings are iterables and are sequences of characters. Hence they can be indexed, sliced, and iterated through, similar to other iterables like lists. Each character has a specific position in the string, which is its index. Using indexes, we can access specific parts of the string. Accessing a single character using a specific position or index in the string is called *indexing* and accessing a part of a string i.e., a substring using a start and end index, is called *slicing*. Python supports two types of indexes—one starting from 0 and increasing by 1 each character until the end of the string, and the other starting from -1 at the end of the string and decreasing by 1 each character until the beginning of the string. Figure 2-8 depicts the two types of indexes for the string, `'PYTHON'`.

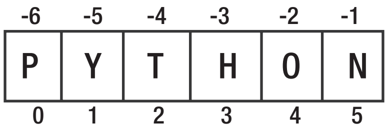


Figure 2-8. *String-indexing syntax*

Let’s get started with some basic string indexing so you can get a feel for how to access specific characters in a string.

```
# creating a string
In [12]: s = 'PYTHON'
    ...: s, type(s)
Out[12]: ('PYTHON', str)
```

```
# depicting string indices
In [13]: for index, character in enumerate(s):
    ...:     print('Character ->', character, 'has index->', index)
Character -> P has index-> 0
Character -> Y has index-> 1
Character -> T has index-> 2
Character -> H has index-> 3
Character -> O has index-> 4
Character -> N has index-> 5
```

```
# string indexing
In [14]: s[0], s[1], s[2], s[3], s[4], s[5]
Out[14]: ('P', 'Y', 'T', 'H', 'O', 'N')
```

```
In [15]: s[-1], s[-2], s[-3], s[-4], s[-5], s[-6]
Out[15]: ('N', 'O', 'H', 'T', 'Y', 'P')
```

It is quite clear that you can access specific string elements with indices similar to how you would access a list. Let's look at some interesting ways to slice strings now with some hands-on examples!

```
# string slicing
In [16]: s[:]
Out[16]: 'PYTHON'
```

```
In [17]: s[1:4]
Out[17]: 'YTH'
```

```
In [18]: s[:3], s[3:]
Out[18]: ('PYT', 'HON')
```

```
In [19]: s[-3:]
Out[19]: 'HON'

In [21]: s[:3] + s[-3:]
Out[21]: 'PYTHON'

In [22]: s[::1]  # no offset
Out[22]: 'PYTHON'

In [24]: s[::2]  # print every 2nd character in string
Out[24]: 'PTO'
```

The preceding snippets should give you some good perspective into how to slice and extract specific substrings from a given string. As mentioned, it is very similar to lists. However, the key difference is that strings are immutable. Let's try to understand this idea in the following examples.

```
# strings are immutable hence assignment throws error
In [27]: s[0] = 'X'

Traceback (most recent call last):

  File "<ipython-input-27-88104b3bc919>", line 1, in <module>
    s[0] = 'X'

TypeError: 'str' object does not support item assignment

# creates a new string
In [28]: print('Original String id:', id(s))
    ...: # creates a new string
    ...: s = 'X' + s[1:]
    ...: print(s)
    ...: print('New String id:', id(s))

Original String id: 2117246774552
XYTHON
New String id: 2117246656048
```

Based on the preceding examples, you can clearly see that strings are immutable and do not support assignment or modifications to the original string in any form. Even if you use the same variable and perform some operations on it, you get a completely new string.

Methods

Strings have a huge arsenal of built-in methods in base Python at your disposal, which you can use for performing various transformations, manipulations, and operations. Discussing each and every method in detail would be out of the current scope; however, this useful link in the official Python documentation https://docs.python.org/3/library/stdtypes.html#string-methods provides all the information you need to know about every method, along with syntax and definition. Methods are extremely useful and increase your productivity, since you do not have to spend extra time writing boilerplate code to handle and manipulate strings. We show some popular examples of string methods in action in the following code snippets.

```
s = 'python is great'

# case conversions
In [33]: s.capitalize()
Out[33]: 'Python is great'

In [34]: s.upper()
Out[34]: 'PYTHON IS GREAT'

In [35]: s.title()
Out[35]: 'Python Is Great'

# string replace
In [36]: s.replace('python', 'NLP')
Out[36]: 'NLP is great'

# Numeric checks
In [37]: '12345'.isdecimal()
Out[37]: True

In [38]: 'apollo11'.isdecimal()
Out[38]: False

# Alphabet checks
In [39]: 'python'.isalpha()
Out[39]: True

In [40]: 'number1'.isalpha()
Out[40]: False
```

```
# Alphanumeric checks
In [41]: 'total'.isalnum()
Out[41]: True

In [42]: 'abc123'.isalnum()
Out[42]: True

In [43]: '1+1'.isalnum()
Out[43]: False
```

The following snippets show some ways to split, join, and strip strings based on different hands-on examples.

```
# String splitting and joining
In [44]: s = 'I,am,a,comma,separated,string'
    ...: s.split(',')
Out[44]: ['I', 'am', 'a', 'comma', 'separated', 'string']

In [45]: ' '.join(s.split(','))
Out[45]: 'I am a comma separated string'

# Basic string stripping
In [46]: s = '   I am surrounded by spaces    '
    ...: s
Out[46]: '   I am surrounded by spaces    '

In [47]: s.strip()
Out[47]: 'I am surrounded by spaces'

# some more combinations
In [48]: sentences = 'Python is great. NLP is also good.'
    ...: sentences.split('.')
Out[48]: ['Python is great', ' NLP is also good', "]

In [49]: print('\n'.join(sentences.split('.')))

Python is great
 NLP is also good
```

```
In [50]: print('\n'.join([sentence.strip()
    ...:                      for sentence in sentences.split('.')
    ...:                          if sentence]))

Python is great
NLP is also good
```

These examples just scratch the surface of the numerous manipulations and operations possible on strings. Feel free to try other operations using different methods mentioned in the docs. We use several of them in subsequent chapters.

Formatting

String formatting is used to substitute specific data objects and types in a string. This is often used when displaying text to the user. There are two different types of formatting used for strings:

- **Formatting expressions:** These expressions are typically of the syntax `'...%s...%s...' %(values)`, where the `%s` denotes a placeholder for substituting a string from the list of strings depicted in values. This is quite similar to the C style `printf` model and has been in Python since the beginning. You can substitute values of other types with the respective alphabet following the `%` symbol, like `%d` is for integers and `%f` for floating point numbers.
- **Formatting methods:** These strings take the form of `'...{}...{}...'.format(values)`, which uses the braces `{}` for placeholders to place strings from values using the `format` method. This was present in Python since the `2.6.x` version.

The following code snippets depict both types of string formatting using several hands-on examples.

```
# Simple string formatting expressions - old style
In [51]: 'Hello %s' %('Python!')
Out[51]: 'Hello Python!'
```

```
In [52]: 'Hello %s %s' %('World!', 'How are you?')
Out[52]: 'Hello World! How are you?'

# Formatting expressions with different data types - old style
In [53]: 'We have %d %s containing %.2f gallons of %s' %(2, 'bottles',
2.5, 'milk')
Out[53]: 'We have 2 bottles containing 2.50 gallons of milk'

In [54]: 'We have %d %s containing %.2f gallons of %s' %(5.21, 'jugs',
10.86763, 'juice')
Out[54]: 'We have 5 jugs containing 10.87 gallons of juice'

# Formatting strings using the format method - new style
In [55]: 'Hello {} {}, it is a great {} to meet you at {}'.format('Mr.',
'Jones', 'pleasure', 5)
Out[55]: 'Hello Mr. Jones, it is a great pleasure to meet you at 5'

In [56]: 'Hello {} {}, it is a great {} to meet you at {} o\'
clock'.format('Sir', 'Arthur', 'honor', 9)
Out[56]: "Hello Sir Arthur, it is a great honor to meet you at 9 o' clock"

# Alternative ways of using string format
In [57]: 'I have a {food_item} and a {drink_item} with me'.format(drink_
item='soda', food_item='sandwich')
Out[57]: 'I have a sandwich and a soda with me'

In [58]: 'The {animal} has the following attributes: {attributes}'.
format(animal='dog', attributes=['lazy', 'loyal'])
Out[58]: "The dog has the following attributes: ['lazy', 'loyal']"
```

From these examples, you can see that there is no hard and fast rule for formatting strings, so go ahead and experiment with different formats and use the one that's best suited to your task. We do recommend going with the new style in general.

Regular Expressions

Regular expressions, known more popularly as *regexes*, allow you to create string patterns and use them for searching and substituting specific pattern matches in textual data. Python offers a rich module named `re` for creating and using regular expressions. Entire books have been written on this topic because it is easy to use but difficult to master. Discussing every aspect of regular expressions would not be possible in the current scope, but we cover the main areas with sufficient examples, which should be enough to get started on this topic.

Regular expressions or regexes are specific patterns often denoted using the raw string notation. These patterns match a specific set of strings based on the rules expressed by the patterns. These patterns then are usually compiled into bytecode, which is then executed for matching strings using a matching engine. The `re` module also provides several flags that can change the way the pattern matches are executed. Some important flags are:

- `re.I` or `re.IGNORECASE` is used to match patterns ignoring case sensitivity.
- `re.S` or `re.DOTALL` causes the period (`.`) character to match any character, including new lines.
- `re.U` or `re.UNICODE` helps match Unicode-based characters (deprecated in Python 3.x).

For pattern matching, there are various rules used in regexes. Some popular ones are:

- `.` for matching a single character
- `^` for matching the start of the string
- `$` for matching the end of the string
- `*` for matching zero or more cases of the previous mentioned regex before the `*` symbol in the pattern
- `?` for matching zero or one case of the previous mentioned regex before the `?` symbol in the pattern
- `[...]` for matching any one of the set of characters inside the square brackets
- `[^...]` for matching a character not present in the square brackets after the `^` symbol

- | denotes the OR operator for matching either the preceding or the next regex
- + for matching one or more cases of the previous mentioned regex before the + symbol in the pattern
- \d for matching decimal digits, which are also depicted as [0-9]
- \D for matching non-digits, also depicted as [^0-9]
- \s for matching whitespace characters
- \S for matching non-whitespace characters
- \w for matching alpha-numeric characters; also depicted as [a-zA-Z0-9_]
- \W for matching non alpha-numeric characters; also depicted as [^a-zA-Z0-9_]

Regular expressions can be compiled into pattern objects and then used with a variety of methods for pattern search and substitution in strings. The main methods offered by the `re` module for performing these operations are as follows:

- `re.compile()`: This method compiles a specified regular expression pattern into a regular expression object, which can be used for matching and searching. Takes a pattern and optional flags as input, which we discussed previously.
- `re.match()`: This method is used to match patterns at the beginning of strings.
- `re.search()`: This method is used to match patterns occurring at any position in the string.
- `re.findall()`: This method returns all non-overlapping matches of the specified regex pattern in the string.
- `re.finditer()`: This method returns all matched instances in the form of an iterator, for a specific pattern in a string when scanned from left to right.
- `re.sub()`: This method is used to substitute a specified regex pattern in a string with a replacement string. It only substitutes the left-most occurrence of the pattern in the string.

The following code snippets depict some of these methods and show how they are typically used when dealing with strings and regular expressions.

```
# creating some strings
s1 = 'Python is an excellent language'
s2 = 'I love the Python language. I also use Python to build applications
at work!'

# due to case mismatch there is no match found
In [61]: import re
    ...:
    ...: pattern = 'python'
    ...: # match only returns a match if regex match is found at the
           beginning of the string
    ...: re.match(pattern, s1)

# pattern is in lower case hence ignore case flag helps in matching same pattern
# with different cases
In [62]: re.match(pattern, s1, flags=re.IGNORECASE)
Out[62]: <_sre.SRE_Match object; span=(0, 6), match='Python'>

# printing matched string and its indices in the original string
In [64]: m = re.match(pattern, s1, flags=re.IGNORECASE)
    ...: print('Found match {} ranging from index {} - {} in the string
         "{}"'.format(m.group(0), m.start(), m.end(), s1))

Found match Python ranging from index 0 - 6 in the string "Python is an
excellent language"

# match does not work when pattern is not there in the beginning of string s2
In [65]: re.match(pattern, s2, re.IGNORECASE)
```

Let's now look at some examples that illustrate how the `find(...)` and `search(...)` methods work in regular expressions:

```
# illustrating find and search methods using the re module
In [66]: re.search(pattern, s2, re.IGNORECASE)
Out[66]: <_sre.SRE_Match object; span=(11, 17), match='Python'>

In [67]: re.findall(pattern, s2, re.IGNORECASE)
Out[67]: ['Python', 'Python']

In [68]: match_objs = re.finditer(pattern, s2, re.IGNORECASE)
    ...: match_objs

Out[68]: <callable_iterator at 0x1ecf5c1c828>

In [69]: print("String:", s2)
    ...: for m in match_objs:
    ...:     print('Found match "{}" ranging from index {} - {}'.format
             (m.group(0), m.start(), m.end()))

String: I love the Python language. I also use Python to build applications
at work!
Found match "Python" ranging from index 11 - 17
Found match "Python" ranging from index 39 - 45
```

Regular expressions for text substitution are useful to find and replace specific text tokens in strings. We illustrate these using a few examples:

```
# illustrating pattern substitution using sub and subn methods
In [81]: re.sub(pattern, 'Java', s2, flags=re.IGNORECASE)
Out[81]: 'I love the Java language. I also use Java to build applications
at work!'

In [82]: re.subn(pattern, 'Java', s2, flags=re.IGNORECASE)
Out[82]: ('I love the Java language. I also use Java to build applications
at work!', 2)
```

```
# dealing with unicode matching using regexes
In [83]: s = u'H\u00e8llo! this is Python 🐍'
    ...: s
Out[83]: 'Hèllo! this is Python 🐍'

In [84]: re.findall(r'\w+', s)
Out[84]: ['Hèllo', 'this', 'is', 'Python']

In [85]: re.findall(r"[A-Z]\w+", s, re.UNICODE)
Out[85]: ['Hèllo', 'Python']

In [86]: emoji_pattern = r"['\U0001F300-\U0001F5FF'|'\U0001F600-
                             \U0001F64F'|'\U0001F680-\U0001F6FF'|'\u2600-
                             \u26FF\u2700-\u27BF']"
    ...: re.findall(emoji_pattern, s, re.UNICODE)

Out[86]: ['🐍']
```

This concludes our discussion of strings and their various aspects, including representation and operations. This should give you an idea of how strings can be utilized for working with text data and how they form the basis for processing text, which is an important component in text analytics. We now cover a basic text processing case study, where we bring everything together based on what we learned in the previous sections.

Basic Text Processing and Analysis: Putting It All Together

Let's utilize what we have learned so far in this chapter as well as the previous chapter to build and solve a basic text-processing problem. For this, we load the King James version of the Bible from the Gutenberg corpus in NLTK. The following code shows us how to load the Bible corpus and display the first few lines in the corpus.

```
from nltk.corpus import gutenberg
import matplotlib.pyplot as plt
% matplotlib inline
```

```
bible = gutenberg.open('bible-kjv.txt')
bible = bible.readlines()
bible[:5]
```

```
['[The King James Bible]\n',
 '\n',
 'The Old Testament of the King James Bible\n',
 '\n',
 'The First Book of Moses:  Called Genesis\n']
```

We can clearly see the first few lines of the Bible corpus in the preceding output. Let's do some basic preprocessing by removing all the empty newlines in our corpus and stripping away any newline characters from other lines.

```
In [88]: len(bible)
Out[88]: 99805
```

```
In [89]: bible = list(filter(None, [item.strip('\n') for item in bible]))
    ...: bible[:5]
```

```
Out[89]:
['[The King James Bible]',
 'The Old Testament of the King James Bible',
 'The First Book of Moses:  Called Genesis',
 '1:1 In the beginning God created the heaven and the earth.',
 '1:2 And the earth was without form, and void; and darkness was upon']
```

```
In [90]: len(bible)
Out[90]: 74645
```

We can clearly see that there were a lot of empty newlines in our corpus and we have been able to successfully remove them. Let's do some basic frequency analysis on our corpus now. Suppose we wanted to visualize the overall distribution of typical sentence or line lengths across the Bible. We can do that by computing the length of each sentence and then visualize this using a histogram, as shown in Figure 2-9.

```
line_lengths = [len(sentence) for sentence in bible]
h = plt.hist(line_lengths)
```

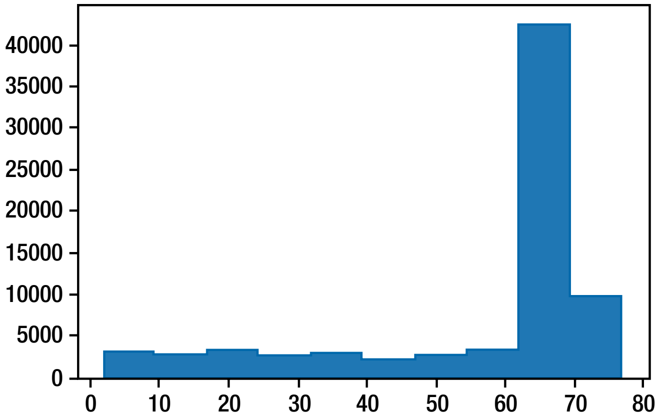


***Figure 2-9.** Visualizing sentence length distributions in the Bible*

Based on the plot depicted in Figure 2-9, it looks like most of the sentences are around 65-70 characters. Let's look at the total words per sentence distribution now. To get that distribution, first let's look at a way to tokenize each sentence in our corpus.

```
In [95]: tokens = [item.split() for item in bible]
    ...: print(tokens[:5])
```

```
[['[The', 'King', 'James', 'Bible]'], ['The', 'Old', 'Testament', 'of',
'the', 'King', 'James', 'Bible'], ['The', 'First', 'Book', 'of', 'Moses:',
'Called', 'Genesis'], ['1:1', 'In', 'the', 'beginning', 'God', 'created',
'the', 'heaven', 'and', 'the', 'earth.'], ['1:2', 'And', 'the', 'earth',
'was', 'without', 'form,', 'and', 'void;', 'and', 'darkness', 'was',
'upon']]
```

Now that we have tokenized each sentence, we just have to compute the length of each sentence to get the total words per sentence and build a histogram to visualize this distribution. See Figure 2-10.

```
In [96]: total_tokens_per_line = [len(sentence.split()) for sentence in bible]
    ...: h = plt.hist(total_tokens_per_line, color='orange')
```

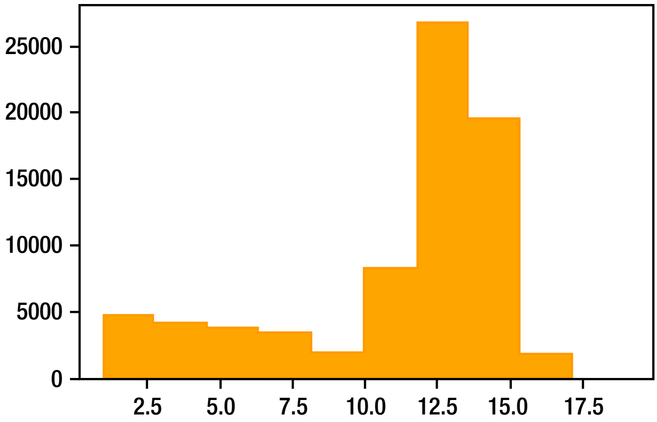


Figure 2-10. *Visualizing total words per sentence distributions in the Bible*

Based on the visualization depicted in Figure 2-10, we can clearly conclude that most sentences in the Bible have roughly 12-15 words, or tokens, in them. Let's now try to determine the most common words in the Bible corpus. We already have our sentences tokenized into words (lists of words). The first step involves flattening this big list of lists (each list is a tokenized sentence of words) into one big list of words.

```
words = [word for sentence in tokens for word in sentence]
print(words[:20])
```

```
['[The', 'King', 'James', 'Bible]', 'The', 'Old', 'Testament', 'of', 'the',
'King', 'James', 'Bible', 'The', 'First', 'Book', 'of', 'Moses:', 'Called',
'Genesis', '1:1']
```

Nice! We have our big list of tokens from our corpus. However, you can see the tokens are not totally clean and we have some unwanted symbols and special characters in some of the words. Let's use the power of regular expressions now to remove them.

```
words = list(filter(None, [re.sub(r'[^A-Za-z]', ", word) for word in
words]))
print(words[:20])
```

```
['The', 'King', 'James', 'Bible', 'The', 'Old', 'Testament', 'of', 'the',
'King', 'James', 'Bible', 'The', 'First', 'Book', 'of', 'Moses', 'Called',
'Genesis', 'In']
```

Based on the regular expression we used in the preceding code, we just removed anything that was not an alphabetical character. Thus all numbers and special characters were removed. We can now determine the most frequent words using the following code.

```
In [99]: from collections import Counter
    ...:
    ...: words = [word.lower() for word in words]
    ...: c = Counter(words)
    ...: c.most_common(10)
```

```
Out[99]:
[('the', 64023),
 ('and', 51696),
 ('of', 34670),
 ('to', 13580),
 ('that', 12912),
 ('in', 12667),
 ('he', 10419),
 ('shall', 9838),
 ('unto', 8997),
 ('for', 8970)]
```

We see a lot of general filler words like pronouns, articles, and so on are the most frequent words, which makes perfect sense. But this doesn't convey much information. What if we could remove these words and focus on the more interesting ones? One approach could be to remove these filler words, popularly known as stopwords, and then compute the frequency as follows.

```
In [100]: import nltk
     ...:
     ...: stopwords = nltk.corpus.stopwords.words('english')
     ...: words = [word.lower() for word in words if word.lower() not in
          stopwords]
     ...: c = Counter(words)
     ...: c.most_common(10)
```

```
Out[100]:
[('shall', 9838),
 ('unto', 8997),
 ('lord', 7830),
 ('thou', 5474),
 ('thy', 4600),
 ('god', 4442),
 ('said', 3999),
 ('ye', 3983),
 ('thee', 3826),
 ('upon', 2748)]
```

Thus, we see that the results are better than before; however, many words are still filler or stopwords. This is more colloquial English, hence they are not a part of the standard English stopwords list so they were not removed. We can always build a custom stopword list as needed. (More on stopwords in Chapter 3.) This should give you a good idea of how we used all aspects pertaining to strings, methods, and transformations to process and analyze text data.

Natural Language Processing Frameworks

We talked about the Python ecosystem being diverse and supporting a wide variety of libraries, frameworks, and modules in diverse domains. Since we will be analyzing textual data and solving several use cases on it, there are dedicated frameworks and libraries for natural language processing and text analytics, which you can just install and start using, just like any other built-in module in the Python standard library. These frameworks have been built over a long period of time and are usually still in active development. Often the way to assess a framework is to see how active its developer community is.

Each framework contains various methods, capabilities, and features for operating on text, getting insights, and making the data ready for further analysis, like applying machine learning algorithms on preprocessed textual data. Leveraging these frameworks saves a lot of effort and time that would have been spent on writing boilerplate code to handle, process, and manipulate text data. Thus, this enables the developers and researchers to focus more on solving the actual problem and the necessary logic and algorithms needed. We have already seen some glimpses of the `nltk` library in the first chapter. The following list of libraries and frameworks are some of the most popular text analytics frameworks and we utilize several of them throughout the course of the book.

- `nltk`: The Natural Language Toolkit is a complete platform that contains over 50 corpora and lexical resources, such as WordNet. Besides this, it also provides the necessary tools, interfaces, and methods to process and analyze text data. The NLTK framework comes with a suite of efficient modules for classification, tokenization, stemming, lemmatization, tagging, parsing, and semantic reasoning. It is the standard workhorse of any NLP project in the industry.
- `pattern`: The `pattern` project gets an honorable mention here since we used it extensively in the first edition of the book. However, due to a lack of official support or a Python 3.x version, we will not be using it in this edition. This started out as a research project at the Computational Linguistics & Psycholinguistics Research Centre at the University of Antwerp. It provides tools and interfaces for web mining, information retrieval, natural language processing, machine learning, and network analysis.
- `spacy`: This is one of the newer libraries relatively as compared to the others but perhaps one of the best libraries for NLP. We can vouch for the fact that spaCy provides industrial-strength natural language processing capabilities by providing the best implementation of each technique and algorithm, which makes NLP tasks efficient in terms of performance and implementation. In fact, spaCy excels at large-scale information extraction tasks. It has been written from the ground up using efficient, memory-managed Cython. Extensive research has also confirmed that spaCy is the fastest in the world. spaCy also works seamlessly with deep learning and machine learning frameworks like TensorFlow, PyTorch, Scikit-Learn, Gensim, and the rest of Python's excellent AI ecosystem. The best part is that spaCy has support for several languages and provides pretrained word vectors!

- `gensim`: The `gensim` library has a rich set of capabilities for semantic analysis, including topic modeling and similarity analysis. But the best part is that it contains a Python port of Google's very popular Word2Vec model (originally available as a C package), which is a neural network model implemented to learn distributed representations of words where similar words (semantic) occur close to each other. Thus, Gensim can be used for semantic analysis as well as feature engineering!
- `textblob`: This is another library that provides several capabilities, including text processing, phrase extraction, classification, POS tagging, text translation, and sentiment analysis. TextBlob makes a lot of difficult things very easy, including language translation and sentiment analysis, by its extremely intuitive and easy to use API.

Besides these, there are several other frameworks and libraries that are not dedicated to text analytics but will be useful when you want to use machine learning or deep learning techniques on textual data. These include the Scikit-Learn, NumPy, and SciPy stack, which are extremely useful for text feature engineering, handling feature sets in the form of matrices, and even performing popular machine learning tasks like similarity computation, text classification, and clustering.

Besides these, deep learning and tensor-based libraries like PyTorch, TensorFlow, and Keras also come in handy if you want to build advanced deep learning models based on deep neural nets, convnets, sequential, and generative models. You can install most of these libraries using the `pip install <library>` command from the command prompt or terminal. We cover any caveats in the upcoming chapters when we use these libraries.

Summary

This chapter provided a bird's-eye yet sufficiently detailed view of the entire Python ecosystem and what the language offers us in terms of its capabilities for handling text data. You have seen the origins behind the Python language and how it has been evolving with time. The language has benefits of being open source, which has resulted in an active developer community constantly striving to improve the language and add new features.

By now, you also know when you should use Python and the drawbacks associated with the language that every developer should keep in mind when building systems and applications. This chapter also provides a clear idea about how to set up our own Python environment and deal with multiple virtual environments. Starting from the very basics, we have taken a deep dive into how to work with text data using the string data type and its various syntaxes, methods, operations, and formats. We have also seen the power of regular expressions and how useful they can be in pattern matching and substitutions.

To conclude our discussion, we looked at various popular text analytics and natural language processing frameworks, which are useful in solving problems and tasks dealing with natural language processing, analyzing, and extracting insights from text data. This should get you started with programming in Python and handling text data. In the next chapter, we build on the foundations of this chapter as we start to understand, process, and parse text data in usable formats.

CHAPTER 3

Processing and Understanding Text

In the previous chapters, we saw a glimpse of the entire natural language processing and text analytics landscape with essential terminology and concepts. Besides this, we were also introduced to the Python programming language, essential constructs, syntax, and learned how to work with strings to manage textual data. To perform complex operations on text with machine learning or deep learning algorithms, you need to process and parse textual data into more easy-to-interpret formats. All machine learning algorithms, be they supervised or unsupervised techniques, work with input features, which are numeric in nature. While this is a separate topic under feature engineering, which we shall explore in detail in the next chapter, to get to that step, you will need to clean, normalize, and preprocess the initial textual data.

The text corpora and other textual data in their native raw formats are normally not well formatted and standardized and of course we should expect this, after all, text data is highly unstructured! Text processing or, to be more specific preprocessing, involves a wide variety of techniques that convert raw text into well-defined sequences of linguistic components that have standard structure and notation. Additional metadata is often also present in the form of annotations to give more meaning to the text components like tags. The following list gives us an idea of some of the most popular text preprocessing and understanding techniques, which we explore in this chapter.

- Removing HTML tags
- Tokenization
- Removing unnecessary tokens and stopwords
- Handling contractions
- Correcting spelling errors

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_3

- Stemming
- Lemmatization
- Tagging
- Chunking
- Parsing

Besides these techniques, you also need to perform some basic operations, like case conversion, dealing with irrelevant components, and removing noise based on the problem to be solved. An important point to remember is that a robust text preprocessing system is always an essential part of any application on NLP and text analytics. The primary reason for that is because all the textual components obtained after preprocessing—be they words, phrases, sentences, or any other tokens—form the basic building blocks of input that's fed into the further stages of the application that perform more complex analyses including learning patterns and extracting information. Hence, the saying "Garbage in Garbage out!" is relevant here because if we do not process the text properly, we will end up getting unwanted and irrelevant results from our applications and systems.

Besides this, text processing helps in cleaning and standardization of the text which not only helps in analytical systems like increasing the accuracy of classifiers but we also get additional information and metadata in the form of annotations. They are very useful in giving more information about the text. We will touch upon normalizing text using various techniques including cleaning, removing unnecessary tokens, stems, and lemmas in this chapter.

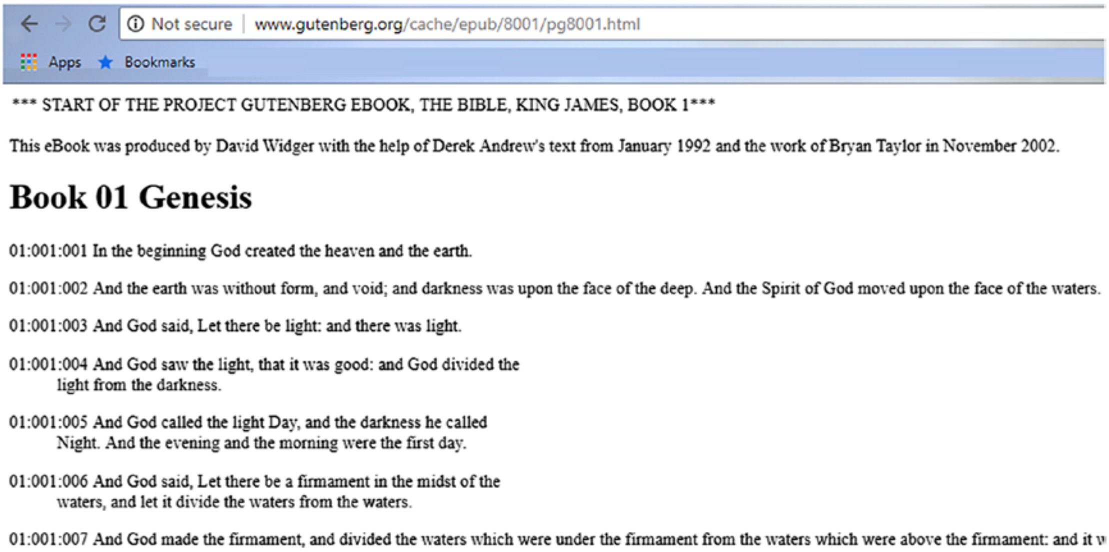
Another important aspect is understanding the textual data after processing and normalizing it. This will involve revisiting some of the concepts surrounding language syntax and structure from Chapter 1, where we talked about sentences, phrases, parts of speech, shallow parsing, and grammar. In this chapter, we look at ways to implement these concepts and use them on real data. We follow a structured and definite path in this chapter, starting from text processing, and gradually exploring the various concepts and techniques associated with it and then moving on to understanding text structure and syntax. Since this book is specifically aimed at practitioners, various code snippets and practical examples also equip you with the right tools and frameworks for implementing these concepts in solving practical problems. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Text Preprocessing and Wrangling

Text *wrangling* (also called *preprocessing* or *normalization*) is a process that consists of a series of steps to wrangle, clean, and standardize textual data into a form that could be consumed by other NLP and intelligent systems powered by machine learning and deep learning. Common techniques for preprocessing include cleaning text, tokenizing text, removing special characters, case conversion, correcting spellings, removing stopwords and other unnecessary terms, stemming, and lemmatization. In this section, we discuss various techniques that are commonly used for text wrangling based on the list we mentioned at the beginning of this chapter. The key idea is to remove unnecessary content from one or more text documents in a corpus (or corpora) and get clean text documents.

Removing HTML Tags

Often, unstructured text contains a lot of noise, especially if you use techniques like web scraping or screen scraping to retrieve data from web pages, blogs, and online repositories. HTML tags, JavaScript, and Iframe tags typically don't add much value to understanding and analyzing text. Our main intent is to extract meaningful textual content from the data extracted from the web. Let's look at a section of a web page showing the King James version of the Bible, freely available thanks to Project Gutenberg, depicted in Figure 3-1.



*** START OF THE PROJECT GUTENBERG EBOOK, THE BIBLE, KING JAMES, BOOK 1***

This eBook was produced by David Widger with the help of Derek Andrew's text from January 1992 and the work of Bryan Taylor in November 2002.

Book 01 Genesis

01:001:001 In the beginning God created the heaven and the earth.

01:001:002 And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters.

01:001:003 And God said, Let there be light: and there was light.

01:001:004 And God saw the light, that it was good: and God divided the
light from the darkness.

01:001:005 And God called the light Day, and the darkness he called
Night. And the evening and the morning were the first day.

01:001:006 And God said, Let there be a firmament in the midst of the
waters, and let it divide the waters from the waters.

01:001:007 And God made the firmament, and divided the waters which were under the firmament from the waters which were above the firmament: and it w

***Figure 3-1.** Section of a web page showing a chapter from the Bible*

We will now leverage requests and retrieve the contents of this web page in Python. This is known as web scraping and the following code helps us achieve this.

```
import requests

data = requests.get('http://www.gutenberg.org/cache/epub/8001/pg8001.html')
content = data.content
print(content[1163:2200])
```

```
b'content="Ebookmaker 0.4.0a5 by Marcello Perathoner <webmaster@gutenberg.
org>" name="generator"/>\r\n</head>\r\n  <body><p id="id00000">Project
Gutenberg EBook The Bible, King James, Book 1: Genesis</p>\r\n\r\n<p
id="id00001">Copyright laws are changing all over the world. Be sure
to check the\r\ncopyright laws for your country before downloading or
redistributing\r\nthis or any other Project Gutenberg eBook.</p>\r\n\r\n<p
id="id00002">This header should be the first thing seen when viewing this
Project\r\nGutenberg file.  Please do not remove it.  Do not change or edit
the\r\nheader without written permission.</p>\r\n\r\n<p id="id00003">Please
read the "legal small print," and other information about the\r\neBook and
Project Gutenberg at the bottom of this file.  Included is\r\nimportant
information about your specific rights and restrictions in\r\nhow the
file may be used.  You can also find out about how to make a\r\ndonation
to Project Gutenberg, and how to get involved.</p>\r\n\r\n<p id="id00004"
style="margin-top: 2em">**Welcome To The World of F'
```

We can clearly see from the preceding output that it is extremely difficult to decipher the actual textual content in the web page, due to all the unnecessary HTML tags. We need to remove those tags. The BeautifulSoup library provides us with some handy functions that help us remove these unnecessary tags with ease.

```
import re
from bs4 import BeautifulSoup

def strip_html_tags(text):
    soup = BeautifulSoup(text, "html.parser")
    [s.extract() for s in soup(['iframe', 'script'])]
    stripped_text = soup.get_text()
    stripped_text = re.sub(r'[\r|\n|\r\n]+', '\n', stripped_text)
    return stripped_text

clean_content = strip_html_tags(content)
print(clean_content[1163:2045])
```

```
*** START OF THE PROJECT GUTENBERG EBOOK, THE BIBLE, KING JAMES, BOOK 1***
This eBook was produced by David Widger
with the help of Derek Andrew's text from January 1992
and the work of Bryan Taylor in November 2002.
Book 01        Genesis
01:001:001 In the beginning God created the heaven and the earth.
01:001:002 And the earth was without form, and void; and darkness was
           upon the face of the deep. And the Spirit of God moved upon
           the face of the waters.
01:001:003 And God said, Let there be light: and there was light.
01:001:004 And God saw the light, that it was good: and God divided the
           light from the darkness.
01:001:005 And God called the light Day, and the darkness he called
           Night. And the evening and the morning were the first day.
01:001:006 And God said, Let there be a firmament in the midst of the
           waters,
```

You can compare this output with the raw web page content and see that we have successfully removed the unnecessary HTML tags. We now have a clean body of text that's easier to interpret and understand.

Text Tokenization

Chapter 1 explained textual structure, its components, and tokens. Tokens are independent and minimal textual components that have some definite syntax and semantics. A paragraph of text or a text document has several components, including sentences, which can be further broken down into clauses, phrases, and words. The most popular tokenization techniques include sentence and word tokenization, which are used to break down a text document (or corpus) into sentences and each sentence into words. Thus, tokenization can be defined as the process of breaking down or splitting textual data into smaller and more meaningful components called *tokens*. In the following sections, we look at some ways to tokenize text into sentences and words.

Sentence Tokenization

Sentence tokenization is the process of splitting a text corpus into sentences that act as the first level of tokens the corpus is comprised of. This is also known as sentence segmentation, since we try to segment the text into meaningful sentences. Any text corpus is a body of text where each paragraph comprises several sentences. There are various ways to perform sentence tokenization. Basic techniques include looking for specific delimiters between sentences like a period (`.`) or a newline character (`\n`) and sometimes even a semicolon (`;`). We will use the NLTK framework, which provides various interfaces for performing sentence tokenization. We primarily focus on the following sentence tokenizers:

- `sent_tokenize`
- Pretrained sentence tokenization models
- `PunktSentenceTokenizer`
- `RegexpTokenizer`

Before we can tokenize sentences, we need some text on which we can try these operations. We load some sample text and part of the Gutenberg corpus available in NLTK. We load the necessary dependencies using the following snippet.

```
import nltk
from nltk.corpus import gutenberg
from pprint import pprint
import numpy as np

# loading text corpora
alice = gutenberg.raw(fileids='carroll-alice.txt')
sample_text = ("US unveils world's most powerful supercomputer, beats China. "
               "The US has unveiled the world's most powerful supercomputer
               called 'Summit', "
               "beating the previous record-holder China's Sunway
               TaihuLight. With a peak performance "
               "of 200,000 trillion calculations per second, it is over
               twice as fast as Sunway TaihuLight, "
               "which is capable of 93,000 trillion calculations per
               second. Summit has 4,608 servers, "
               "which reportedly take up the size of two tennis courts.")
sample_text
```

```
"US unveils world's most powerful supercomputer, beats China. The US
has unveiled the world's most powerful supercomputer called 'Summit',
beating the previous record-holder China's Sunway TaihuLight. With a
peak performance of 200,000 trillion calculations per second, it is over
twice as fast as Sunway TaihuLight, which is capable of 93,000 trillion
calculations per second. Summit has 4,608 servers, which reportedly take up
the size of two tennis courts."
```

We can check the length of the "Alice in Wonderland" corpus and the first few lines in it using the following snippet.

```
# Total characters in Alice in Wonderland
len(alice)
```

```
144395
```

```
# First 100 characters in the corpus
alice[0:100]
```

```
"[Alice's Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER
I. Down the Rabbit-Hole\n\nAlice was"
```

Default Sentence Tokenizer

The `nltk.sent_tokenize(...)` function is the default sentence tokenization function that NLTK recommends and it uses an instance of the `PunktSentenceTokenizer` class internally. However, this is not just a normal object or instance of that class. It has been pretrained on several language models and works really well on many popular languages besides English. The following snippet shows the basic usage of this function on our text samples.

```
default_st = nltk.sent_tokenize
alice_sentences = default_st(text=alice)
sample_sentences = default_st(text=sample_text)

print('Total sentences in sample_text:', len(sample_sentences))
print('Sample text sentences :-')
print(np.array(sample_sentences))

print('\nTotal sentences in alice:', len(alice_sentences))
print('First 5 sentences in alice:-')
print(np.array(alice_sentences[0:5]))
```

Upon running this snippet, you get the following output depicting the total number of sentences and what those sentences look like in the text corpora.

```
Total sentences in sample_text: 4
Sample text sentences :-
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']

Total sentences in alice: 1625
First 5 sentences in alice:-
["[Alice's Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER I."
 "Down the Rabbit-Hole\n\nAlice was beginning to get very tired of sitting
by her sister on the\nbank, and of having nothing to do: once or twice she
had peeped into the\nbook her sister was reading, but it had no pictures
or conversations in\nit, 'and what is the use of a book,' thought Alice
'without pictures or\nconversation?'"
 'So she was considering in her own mind (as well as she could, for the\nhot
day made her feel very sleepy and stupid), whether the pleasure\nof making
a daisy-chain would be worth the trouble of getting up and\npicking the
daisies, when suddenly a White Rabbit with pink eyes ran\nclose by her.'
```

```
 "There was nothing so VERY remarkable in that; nor did Alice think it so\
nVERY much out of the way to hear the Rabbit say to itself, 'Oh dear!"
 'Oh dear!']
```

Now, as you can see, the tokenizer is quite intelligent. It doesn't just use periods to delimit sentences, but also considers other punctuation and capitalization of words. We can also tokenize text of other languages using some pretrained models present in NLTK.

Pretrained Sentence Tokenizer Models

Suppose we were dealing with German text. We can use `sent_tokenize`, which is already trained, or load a pretrained tokenization model on German text into a `PunktSentenceTokenizer` instance and perform the same operation. The following snippet shows this. We start by loading a German text corpus and inspecting it.

```
from nltk.corpus import europarl_raw

german_text = europarl_raw.german.raw(fileids='ep-00-01-17.de')
# Total characters in the corpus
print(len(german_text))
# First 100 characters in the corpus
print(german_text[0:100])

157171

Wiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem 17.
Dezember unterbrochene Sit
```

Next, we tokenize the text corpus into sentences using the default `sent_tokenize(...)` tokenizer and a pretrained German language tokenizer by loading it from the NLTK resources.

```
# default sentence tokenizer
german_sentences_def = default_st(text=german_text, language='german')

# loading german text tokenizer into a PunktSentenceTokenizer instance
german_tokenizer = nltk.data.load(resource_url='tokenizers/punkt/german.
pickle')
german_sentences = german_tokenizer.tokenize(german_text)
```

We can now verify the time of our German tokenizer and check if the results obtained by using the two tokenizers match!

```
# verify the type of german_tokenizer
# should be PunktSentenceTokenizer
print(type(german_tokenizer))
```

```
<class 'nltk.tokenize.punkt.PunktSentenceTokenizer'>
```

```
# check if results of both tokenizers match
# should be True
print(german_sentences_def == german_sentences)
```

```
True
```

Thus we see that indeed the `german_tokenizer` is an instance of `PunktSentenceTokenizer`, which specializes in dealing with the German language. We also checked if the sentences obtained from the default tokenizer are the same as the sentences obtained by this pretrained tokenizer. As expected, they are the same (`true`). We also print some sample tokenized sentences from the output.

```
# print first 5 sentences of the corpus
print(np.array(german_sentences[:5]))
```

```
[' \nWiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem
17. Dezember unterbrochene Sitzungsperiode des Europäischen Parlaments für
wiederaufgenommen , wünsche Ihnen nochmals alles Gute zum Jahreswechsel und
hoffe , daß Sie schöne Ferien hatten .'
 'Wie Sie feststellen konnten , ist der gefürchtete " Millenium-Bug " nicht
eingetreten .'
 'Doch sind Bürger einiger unserer Mitgliedstaaten Opfer von schrecklichen
Naturkatastrophen geworden .'
 'Im Parlament besteht der Wunsch nach einer Aussprache im Verlauf dieser
Sitzungsperiode in den nächsten Tagen .'
 'Heute möchte ich Sie bitten - das ist auch der Wunsch einiger
Kolleginnen und Kollegen - , allen Opfern der Stürme , insbesondere in den
verschiedenen Ländern der Europäischen Union , in einer Schweigeminute zu
gedenken .']
```

Thus we see that our assumption was indeed correct and you can tokenize sentences belonging to different languages in two different ways.

PunktSentenceTokenizer

Using the default `PunktSentenceTokenizer` class is also pretty straightforward, as the following snippet shows.

```
punkt_st = nltk.tokenize.PunktSentenceTokenizer()
sample_sentences = punkt_st.tokenize(sample_text)
print(np.array(sample_sentences))
```

```
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']
```

RegexpTokenizer

The last tokenizer we cover in sentence tokenization is using an instance of the `RegexpTokenizer` class to tokenize text into sentences, where we will use specific regular expression-based patterns to segment sentences. Recall the regular expressions from the previous chapter if you want to refresh your memory. The following snippet shows how to use a regex pattern to tokenize sentences.

```
SENTENCE_TOKENS_PATTERN = r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<![A-Z]\.)
(?<=\.|\?|\!)\s'
regex_st = nltk.tokenize.RegexpTokenizer(
            pattern=SENTENCE_TOKENS_PATTERN,
            gaps=True)
sample_sentences = regex_st.tokenize(sample_text)
print(np.array(sample_sentences))
```

```
["US unveils world's most powerful supercomputer, beats China."
 "The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight."
 'With a peak performance of 200,000 trillion calculations per second, it
is over twice as fast as Sunway TaihuLight, which is capable of 93,000
trillion calculations per second.'
 'Summit has 4,608 servers, which reportedly take up the size of two tennis
courts.']
```

This output shows that we obtained the same sentences as we had obtained using the other tokenizers. This gives us an idea of tokenizing text into sentences using different NLTK interfaces. In the following section, we look at tokenizing these sentences into words using several techniques.

Word Tokenization

Word tokenization is the process of splitting or segmenting sentences into their constituent words. A sentence is a collection of words and with tokenization we essentially split a sentence into a list of words that can be used to reconstruct the sentence. Word tokenization is really important in many processes, especially in cleaning and normalizing text where operations like stemming and lemmatization work on each individual word based on its respective stems and lemma. Similar to sentence tokenization, NLTK provides various useful interfaces for word tokenization. We will touch up on the following main interfaces:

- `word_tokenize`
- `TreebankWordTokenizer`
- `TokTokTokenizer`
- `RegexpTokenizer`
- Inherited tokenizers from `RegexpTokenizer`

We leverage our sample text data from the previous section to demonstrate hands-on examples.

Default Word Tokenizer

The nltk.word_tokenize(...) function is the default and recommended word tokenizer, as specified by NLTK. This tokenizer is an instance or object of the TreebankWordTokenizer class in its internal implementation and acts as a wrapper to that core class. The following snippet illustrates its usage.

```
default_wt = nltk.word_tokenize
words = default_wt(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway', 'TaihuLight',
       '.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

TreebankWordTokenizer

The TreebankWordTokenizer is based on the Penn Treebank and uses various regular expressions to tokenize the text. Of course, one primary assumption here is that we have already performed sentence tokenization beforehand. The original tokenizer used in the Penn Treebank is available as a sed script and you can check it out at http://www.cis.upenn.edu/~treebank/tokenizer.sed to get an idea of the patterns used to tokenize the sentences into words. Some of the main features of this tokenizer are mentioned here:

- Splits and separates out periods that appear at the end of a sentence
- Splits and separates commas and single quotes when followed by whitespace

- Most punctuation characters are split and separated into independent tokens
- Splits words with standard contractions, such as don't to do and n't

The following snippet shows the usage of the TreebankWordTokenizer for word tokenization.

```
treebank_wt = nltk.TreebankWordTokenizer()
words = treebank_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway',
       'TaihuLight.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

As expected, the output is similar to word_tokenize(), since they use the same tokenizing mechanism.

TokTokTokenizer

TokTokTokenizer is one of the newer tokenizers introduced by NLTK present in the nltk.tokenize.toktok module. In general, the tok-tok tokenizer is a general tokenizer, where it assumes that the input has one sentence per line. Hence, only the final period is tokenized. However, as needed, we can remove the other periods from the words using regular expressions. Tok-tok has been tested on, and gives reasonably good results for, English, Persian, Russian, Czech, French, German, Vietnamese, and many other languages. It is in fact a Python port of https://github.com/jonsafari/tok-tok, where there is also a Perl implementation. The following code shows a tokenization operation using the TokTokTokenizer.

```
from nltk.tokenize.toktok import TokTokTokenizer
tokenizer = ToktokTokenizer()
words = tokenizer.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "'", ',', 'beating',
       'the', 'previous', 'record-holder', 'China', "'", 's', 'Sunway',
       'TaihuLight.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

RegexpTokenizer

We now look at how to use regular expressions and the RegexpTokenizer class to tokenize sentences into words. Remember that there are two main parameters that are useful in tokenization—the regex pattern for building the tokenizer and the gaps parameter, which, if set to true, is used to find the gaps between the tokens. Otherwise, it is used to find the tokens themselves. The following code snippet shows some examples of using regular expressions to perform word tokenization.

```
# pattern to identify tokens themselves
TOKEN_PATTERN = r'\w+'
regex_wt = nltk.RegexpTokenizer(pattern=TOKEN_PATTERN,
                                gaps=False)
words = regex_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', 's', 'most', 'powerful', 'supercomputer',
       'beats', 'China', 'The', 'US', 'has', 'unveiled', 'the', 'world',
       's', 'most', 'powerful', 'supercomputer', 'called', 'Summit',
       'beating', 'the', 'previous', 'record', 'holder', 'China', 's',
       'Sunway', 'TaihuLight', 'With', 'a', 'peak', 'performance', 'of',
       '200', '000', 'trillion', 'calculations', 'per', 'second', 'it',
       'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
       'which', 'is', 'capable', 'of', '93', '000', 'trillion',
       'calculations', 'per', 'second', 'Summit', 'has', '4', '608',
       'servers', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts'], dtype='<U13')
```

```
# pattern to identify tokens by using gaps between tokens
GAP_PATTERN = r'\s+'
regex_wt = nltk.RegexpTokenizer(pattern=GAP_PATTERN,
                                gaps=True)
words = regex_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', "world's", 'most', 'powerful', 'supercomputer,',
       'beats', 'China.', 'The', 'US', 'has', 'unveiled', 'the',
       "world's", 'most', 'powerful', 'supercomputer', 'called',
       "'Summit',", 'beating', 'the', 'previous', 'record-holder',
       "China's", 'Sunway', 'TaihuLight.', 'With', 'a', 'peak',
       'performance', 'of', '200,000', 'trillion', 'calculations', 'per',
       'second,', 'it', 'is', 'over', 'twice', 'as', 'fast', 'as',
       'Sunway', 'TaihuLight,', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second.', 'Summit', 'has',
       '4,608', 'servers,', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts.'], dtype='<U14')
```

Thus, you can see that there are multiple ways of obtaining the same results leveraging token patterns themselves or gap patterns. The following code shows us how to obtain the token boundaries for each token during the tokenize operation.

```
word_indices = list(regex_wt.span_tokenize(sample_text))
print(word_indices)
print(np.array([sample_text[start:end] for start, end in word_indices]))
```

```
[(0, 2), (3, 10), (11, 18), (19, 23), (24, 32), (33, 47), (48, 53), (54,
60), (61, 64), (65, 67), (68, 71), (72, 80), (81, 84), (85, 92), (93, 97),
(98, 106), (107, 120), (121, 127), (128, 137), (138, 145), (146, 149),
(150, 158), (159, 172), (173, 180), (181, 187), (188, 199), (200, 204),
(205, 206), (207, 211), (212, 223), (224, 226), (227, 234), (235, 243),
(244, 256), (257, 260), (261, 268), (269, 271), (272, 274), (275, 279),
(280, 285), (286, 288), (289, 293), (294, 296), (297, 303), (304, 315),
(316, 321), (322, 324), (325, 332), (333, 335), (336, 342), (343, 351),
(352, 364), (365, 368), (369, 376), (377, 383), (384, 387), (388, 393),
(394, 402), (403, 408), (409, 419), (420, 424), (425, 427), (428, 431),
(432, 436), (437, 439), (440, 443), (444, 450), (451, 458)]
['US' 'unveils' "world's" 'most' 'powerful' 'supercomputer,' 'beats'
 'China.' 'The' 'US' 'has' 'unveiled' 'the' "world's" 'most' 'powerful'
 'supercomputer' 'called' "'Summit'," 'beating' 'the' 'previous'
 'record-holder' "China's" 'Sunway' 'TaihuLight.' 'With' 'a' 'peak'
 'performance' 'of' '200,000' 'trillion' 'calculations' 'per' 'second,'
 'it' 'is' 'over' 'twice' 'as' 'fast' 'as' 'Sunway' 'TaihuLight,' 'which'
 'is' 'capable' 'of' '93,000' 'trillion' 'calculations' 'per' 'second.'
 'Summit' 'has' '4,608' 'servers,' 'which' 'reportedly' 'take' 'up' 'the'
 'size' 'of' 'two' 'tennis' 'courts.']
```

Inherited Tokenizers from RegexpTokenizer

Besides the base RegexpTokenizer class, there are several derived classes that perform different types of word tokenization. The WordPunktTokenizer uses the pattern r'\w+|[^\w\s]+' to tokenize sentences into independent alphabetic and non-alphabetic tokens.

```
wordpunkt_wt = nltk.WordPunctTokenizer()
words = wordpunkt_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'", 's', 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "',", 'beating', 'the',
       'previous', 'record', '-', 'holder', 'China', "'", 's', 'Sunway',
       'TaihuLight', '.', 'With', 'a', 'peak', 'performance', 'of', '200',
       ',', '000', 'trillion', 'calculations', 'per', 'second', ',', 'it',
       'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
       ',', 'which', 'is', 'capable', 'of', '93', ',', '000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4', ',',
       '608', 'servers', ',', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

The WhitespaceTokenizer tokenizes sentences into words based on whitespace, like tabs, newlines, and spaces. The following snippet shows demonstrations of these tokenizers.

```
whitespace_wt = nltk.WhitespaceTokenizer()
words = whitespace_wt.tokenize(sample_text)
np.array(words)
```

```
array(['US', 'unveils', "world's", 'most', 'powerful', 'supercomputer,',
       'beats', 'China.', 'The', 'US', 'has', 'unveiled', 'the',
       "world's", 'most', 'powerful', 'supercomputer', 'called',
       "'Summit',", 'beating', 'the', 'previous', 'record-holder',
       "China's", 'Sunway', 'TaihuLight.', 'With', 'a', 'peak',
       'performance', 'of', '200,000', 'trillion', 'calculations', 'per',
       'second,', 'it', 'is', 'over', 'twice', 'as', 'fast', 'as',
       'Sunway', 'TaihuLight,', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second.', 'Summit', 'has',
       '4,608', 'servers,', 'which', 'reportedly', 'take', 'up', 'the',
       'size', 'of', 'two', 'tennis', 'courts.'], dtype='<U14')
```

Building Robust Tokenizers with NLTK and spaCy

For a typical NLP pipeline, I recommend leveraging state-of-the-art libraries like NLTK and spaCy and using some of their robust utilities to build a custom function to perform both sentence- and word-level tokenization. A simple example is depicted in the following snippets. We start with looking at how we can leverage NLTK.

```
def tokenize_text(text):
    sentences = nltk.sent_tokenize(text)
    word_tokens = [nltk.word_tokenize(sentence) for sentence in sentences]
    return word_tokens
```

```
sents = tokenize_text(sample_text)
np.array(sents)
```

```
array([list(['US', 'unveils', 'world', "'s", 'most', 'powerful',
             'supercomputer', ',', 'beats', 'China', '.']),
       list(['The', 'US', 'has', 'unveiled', 'the', 'world', "'s", 'most',
             'powerful', 'supercomputer', 'called', "'Summit", "'", ',',
             'beating', 'the', 'previous', 'record-holder', 'China', "'s",
             'Sunway', 'TaihuLight', '.']),
       list(['With', 'a', 'peak', 'performance', 'of', '200,000',
             'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
             'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
             ',', 'which', 'is', 'capable', 'of', '93,000', 'trillion',
             'calculations', 'per', 'second', '.']),
       list(['Summit', 'has', '4,608', 'servers', ',', 'which',
             'reportedly', 'take', 'up', 'the', 'size', 'of', 'two',
             'tennis', 'courts', '.'])], dtype=object)
```

We can also get to the level of word-level tokenization by leveraging list comprehensions, as depicted in the following code.

```
words = [word for sentence in sents for word in sentence]
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'Summit", "'", ',', 'beating', 'the',
       'previous', 'record-holder', 'China', "'s", 'Sunway', 'TaihuLight',
       '.', 'With', 'a', 'peak', 'performance', 'of', '200,000',
       'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
       'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight', ',',
       'which', 'is', 'capable', 'of', '93,000', 'trillion',
       'calculations', 'per', 'second', '.', 'Summit', 'has', '4,608',
       'servers', ',', 'which', 'reportedly', 'take', 'up', 'the', 'size',
       'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

In a similar way, we can leverage spaCy to perform sentence- and word-level tokenizations really quickly, as depicted in the following snippets.

```
import spacy
nlp = spacy.load('en_core', parse = True, tag=True, entity=True)
text_spacy = nlp(sample_text)

sents = np.array(list(text_spacy.sents))
sents

array([US unveils world's most powerful supercomputer, beats China.,
       The US has unveiled the world's most powerful supercomputer called
       'Summit', beating the previous record-holder China's Sunway TaihuLight.,
       With a peak performance of 200,000 trillion calculations per second,
       it is over twice as fast as Sunway TaihuLight, which is capable of
       93,000 trillion calculations per second.,
       Summit has 4,608 servers, which reportedly take up the size of two
       tennis courts.],
      dtype=object)

sent_words = [[word.text for word in sent] for sent in sents]
np.array(sent_words)

array([list(['US', 'unveils', 'world', "'s", 'most', 'powerful',
             'supercomputer', ',', 'beats', 'China', '.']),
       list(['The', 'US', 'has', 'unveiled', 'the', 'world', "'s", 'most',
             'powerful', 'supercomputer', 'called', "'", 'Summit', "'",
             ',', 'beating', 'the', 'previous', 'record', '-', 'holder',
             'China', "'s", 'Sunway', 'TaihuLight', '.']),
       list(['With', 'a', 'peak', 'performance', 'of', '200,000',
             'trillion', 'calculations', 'per', 'second', ',', 'it', 'is',
             'over', 'twice', 'as', 'fast', 'as', 'Sunway', 'TaihuLight',
             ',', 'which', 'is', 'capable', 'of', '93,000', 'trillion',
             'calculations', 'per', 'second', '.']),
       list(['Summit', 'has', '4,608', 'servers', ',', 'which',
             'reportedly', 'take', 'up', 'the', 'size', 'of', 'two',
             'tennis', 'courts', '.'])],
      dtype=object)
```

```
words = [word.text for word in text_spacy]
np.array(words)
```

```
array(['US', 'unveils', 'world', "'s", 'most', 'powerful',
       'supercomputer', ',', 'beats', 'China', '.', 'The', 'US', 'has',
       'unveiled', 'the', 'world', "'s", 'most', 'powerful',
       'supercomputer', 'called', "'", 'Summit', "'", ',', 'beating',
       'the', 'previous', 'record', '-', 'holder', 'China', "'s",
       'Sunway', 'TaihuLight', '.', 'With', 'a', 'peak', 'performance',
       'of', '200,000', 'trillion', 'calculations', 'per', 'second', ',',
       'it', 'is', 'over', 'twice', 'as', 'fast', 'as', 'Sunway',
       'TaihuLight', ',', 'which', 'is', 'capable', 'of', '93,000',
       'trillion', 'calculations', 'per', 'second', '.', 'Summit', 'has',
       '4,608', 'servers', ',', 'which', 'reportedly', 'take', 'up',
       'the', 'size', 'of', 'two', 'tennis', 'courts', '.'], dtype='<U13')
```

This should be more than enough to get you started with text tokenization. We encourage you to play around with more text data and see if you can make it even better!

Removing Accented Characters

Usually in any text corpus, you might be dealing with accented characters/letters, especially if you only want to analyze the English language. Hence, we need to make sure that these characters are converted and standardized into ASCII characters. This shows a simple example—converting **é** to **e**. The following function is a simple way of tackling this task.

```
import unicodedata

def remove_accented_chars(text):
    text = unicodedata.normalize('NFKD', text).encode('ascii', 'ignore').
    decode('utf-8', 'ignore')
    return text

remove_accented_chars('Sómě Áccěntěd těxt')
```

```
'Some Accented text'
```

The preceding function shows us how we can easily convert accented characters to normal English characters, which helps standardize the words in our corpus.

Expanding Contractions

Contractions are shortened versions of words or syllables. These exist in written and spoken forms. Shortened versions of existing words are created by removing specific letters and sounds. In the case of English contractions, they are often created by removing one of the vowels from the word. Examples include “is not” to “isn’t” and “will not” to “won’t”, where you can notice the apostrophe being used to denote the contraction and some of the vowels and other letters being removed.

Contractions are often avoided when in formal writing, but are used quite extensively in informal communication. Various forms of contractions exist and they are tied to the type of auxiliary verbs, which give us normal contractions, negated contractions, and other special colloquial contractions, some of which may not involve auxiliaries.

By nature, contractions pose a problem for NLP and text analytics because, to start with, we have a special apostrophe character in the word. Besides this, we also have two or more words represented by a contraction and this opens a whole new can of worms when we try to tokenize them or standardize the words. Hence, there should be some definite process for dealing with contractions when processing text.

Ideally, you can have a proper mapping for contractions and their corresponding expansions and then use that to expand all the contractions in your text. I have created a vocabulary for contractions and their corresponding expanded forms, which you can access in the file named `contractions.py` in a Python dictionary (available along with the code files for this chapter). A part of the contractions dictionary is shown in the following snippet.

```
CONTRACTION_MAP = {
    "ain't": "is not",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
     .
     .
     .
    "you'll've": "you will have",
    "you're": "you are",
    "you've": "you have"
}
```

Remember, however, that some of the contractions can have multiple forms, such the contraction “you’ll” which can be either “you will” or “you shall”. To make things simple here, we use only one of these expanded forms for each contraction. For our next step, to expand contractions, we use the following code snippet.

```
from contractions import CONTRACTION_MAP
import re

def expand_contractions(text, contraction_mapping=CONTRACTION_MAP):

    contractions_pattern = re.compile('({})'.format('|'.join(contraction_
    mapping.keys())), flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                                if contraction_mapping.get(match)\
                                else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

In this snippet, we use the expanded_match function inside the main expand_contractions function to find each contraction that matches the regex pattern we create out of all the contractions in our CONTRACTION_MAP dictionary. On matching any contraction, we substitute it with its corresponding expanded version and retain the correct case of the word. Let’s see this process in action now!

```
expand_contractions("Y'all can't expand contractions I'd think")

'You all cannot expand contractions I would think'
```

We can see how our function helps expand the contractions from the preceding output. Are there better ways of doing this? Definitely! If we have enough examples, we can even train a deep learning model for better performance.

Removing Special Characters

Special characters and symbols are usually non-alphanumeric characters or even occasionally numeric characters (depending on the problem), which add to the extra noise in unstructured text. Usually, simple regular expressions (regexes) can be used to remove them. The following code helps us remove special characters.

```
def remove_special_characters(text, remove_digits=False):
    pattern = r'[^a-zA-z0-9\s]' if not remove_digits else r'[^a-zA-z\s]'
    text = re.sub(pattern, '', text)
    return text
```

```
remove_special_characters("Well this was fun! What do you think? 123#@!",
                          remove_digits=True)
```

```
'Well this was fun What do you think '
```

I've kept removing digits optional, because often we might need to keep them in the preprocessed text.

Case Conversions

Often you might want to modify the case of words or sentences to make things easier, like matching specific words or tokens. Usually, there are two types of case conversion operations that are used a lot. These are lower- and uppercase conversions, where a body of text is converted completely to lowercase or uppercase. There are other forms also like sentence case or title case. Lowercase is a form where all the letters of the text are small letters and in uppercase they are all capitalized. Title case will capitalize the first letter of each word in the sentence. The following snippet illustrates these concepts.

```
# lowercase
text = 'The quick brown fox jumped over The Big Dog'
text.lower()
```

```
'the quick brown fox jumped over the big dog'
```

```
# uppercase
text.upper()
```

```
'THE QUICK BROWN FOX JUMPED OVER THE BIG DOG'
```

```
# title case
text.title()
```

```
'The Quick Brown Fox Jumped Over The Big Dog'
```

Text Correction

One of the main challenges faced in text wrangling is the presence of incorrect words in the text. The definition of *incorrect* here covers words that have spelling mistakes as well as words with several letters repeated that do not contribute much to its overall significance. To illustrate some examples, the word "finally" could be mistakenly written as "fianlly" or someone expressing intense emotion could write it as "finalllllyyyyyy". The main objective here is to standardize different forms of these words to the correct form so that we do not end up losing vital information from different tokens in the text. We cover dealing with repeated characters as well as correcting spellings in this section.

Correcting Repeating Characters

We just mentioned words that often contain several repeating characters that could be due to incorrect spellings, slang language, or even people wanting to express strong emotions. We show a method here that uses a combination of syntax and semantics to correct these words. We start by correcting the syntax of these words and then move on to semantics.

The first step in our algorithm is to identify repeated characters in a word using a regex pattern and then use a substitution to remove the characters one by one. Let's consider the word "finalllyyy" from the earlier example. The pattern `r'(\w*)(\w)\2(\w*)'` can be used to identify characters that occur twice among other characters in the word. In each step, we try to eliminate one of the repeated characters using a substitution for the match by utilizing the regex match groups (groups 1, 2, and 3) using the pattern `r'\1\2\3'`. Then we keep iterating through this process until no repeated characters remain. The following snippet illustrates this process.

```
old_word = 'finalllyyy'
repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
match_substitution = r'\1\2\3'
step = 1
```

```
while True:
    # remove one repeated character
    new_word = repeat_pattern.sub(match_substitution,
                                  old_word)
    if new_word != old_word:
         print('Step: {} Word: {}'.format(step, new_word))
         step += 1 # update step
         # update old word to last substituted state
         old_word = new_word
         continue
    else:
         print("Final word:", new_word)
         break
```

```
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Step: 4 Word: finaly
Final word: finaly
```

This snippet shows us how one repeated character is removed at each stage and we end up with the word “finaly” in the end. However, this word is incorrect and the correct word was “finally,” which we had obtained in Step 3. We will now utilize the WordNet corpus to check for valid words at each stage and terminate the loop once it is obtained. This introduces the semantic correction needed for our algorithm, as illustrated in the following snippet.

```
from nltk.corpus import wordnet
old_word = 'finalllyyy'
repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
match_substitution = r'\1\2\3'
step = 1

while True:
    # check for semantically correct word
    if wordnet.synsets(old_word):
        print("Final correct word:", old_word)
        break
    # remove one repeated character
    new_word = repeat_pattern.sub(match_substitution,
                                  old_word)
    if new_word != old_word:
        print('Step: {} Word: {}'.format(step, new_word))
        step += 1 # update step
        # update old word to last substituted state
        old_word = new_word
        continue
    else:
        print("Final word:", new_word)
        break
```

```
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Final correct word: finally
```

Thus, we see from this snippet that the code correctly terminated after the third step and we obtained the correct word adhering to both syntax and semantics. We can build a better version of this code by writing the logic in a function, as depicted here, to make it more generic to deal with incorrect tokens from a list of tokens.

```
from nltk.corpus import wordnet

def remove_repeated_characters(tokens):
    repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
    match_substitution = r'\1\2\3'
    def replace(old_word):
        if wordnet.synsets(old_word):
            return old_word
        new_word = repeat_pattern.sub(match_substitution, old_word)
        return replace(new_word) if new_word != old_word else new_word

    correct_tokens = [replace(word) for word in tokens]
    return correct_tokens
```

In this snippet, we use the inner function `replace()` to basically emulate the behavior of our algorithm that we illustrated earlier and then call it repeatedly on each token in a sentence in the outer function `remove_repeated_characters()`. We can see the code in action in the following snippet with an example sentence.

```
sample_sentence = 'My schooool is realllllyyy amaaazingggg'
correct_tokens = remove_repeated_characters(nltk.word_tokenize(sample_
sentence))
' '.join(correct_tokens)
```

```
'My school is really amazing'
```

We can see from this output that our function performs as intended and replaces the repeating characters in each token, giving us correct tokens as desired.

Correcting Spellings

The second problem we face with words is incorrect or wrong spellings that occur due to human error and even machine based errors, which you might have seen with features like auto-correcting text. There are various ways to deal with incorrect spellings where the final objective is to have tokens of text with the correct spelling. We will talk about one of the famous algorithms developed by Peter Norvig, the director of research at Google. You can find the complete detailed post explaining his algorithm and findings at http://norvig.com/spell-correct.html, which we will be exploring in this section.

The main objective of this exercise is that given a problematic word, we need to find the most likely correct form of that word. The approach we follow is to generate a set of candidate words that are near to our input word and select the most likely word from this set as the correct word. We use a corpus of correct English words in this context to identify the correct word based on its frequency in the corpus from our final set of candidates with the nearest distance to our input word. This distance measures how near or far a word is from our input word and is also called the *edit distance*.

The input corpus we use is a file with several books from the Gutenberg corpus and a list of most frequent words from Wiktionary and the British National Corpus. You can find the file under the name `big.txt` in this chapter's code resources or you can download it from Norvig's direct link at http://norvig.com/big.txt and use it. We use the following code snippet to generate a map of frequently occurring words in the English language and their counts.

```
import re, collections

def tokens(text):
    """
    Get all words from the corpus
    """
    return re.findall('[a-z]+', text.lower())

WORDS = tokens(open('big.txt').read())
WORD_COUNTS = collections.Counter(WORDS)
# top 10 words in corpus
WORD_COUNTS.most_common(10)

[('the', 80030), ('of', 40025), ('and', 38313), ('to', 28766), ('in', 22050),
 ('a', 21155), ('that', 12512), ('he', 12401), ('was', 11410), ('it', 10681)]
```

Once we have our vocabulary, we define three functions that compute sets of words that are zero, one, and two edits away from our input word. These edits can be made by the means of insertions, deletions, additions, and transpositions. The following code defines the functions.

```
def edits0(word):
    """
    Return all strings that are zero edits away
    from the input word (i.e., the word itself).
    """
    return {word}

def edits1(word):
    """
    Return all strings that are one edit away
    from the input word.
    """
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    def splits(word):
        """
        Return a list of all possible (first, rest) pairs
        that the input word is made of.
        """
        return [(word[:i], word[:i])
                for i in range(len(word)+1)]

    pairs      = splits(word)
    deletes    = [a+b[1:]           for (a, b) in pairs if b]
    transposes = [a+b[1]+b[0]+b[2:] for (a, b) in pairs if len(b) > 1]
    replaces   = [a+c+b[1:]         for (a, b) in pairs for c in alphabet if b]
    inserts    = [a+c+b             for (a, b) in pairs for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    """Return all strings that are two edits away
    from the input word.
    """
    return {e2 for e1 in edits1(word) for e2 in edits1(e1)}
```

We also define a function called known(), which returns a subset of words from our candidate set of words obtained from the edit functions based on whether they occur in our vocabulary dictionary WORD_COUNTS. This gives us a list of valid words from our set of candidate words.

```
def known(words):
    """
    Return the subset of words that are actually
    in our WORD_COUNTS dictionary.
    """
    return {w for w in words if w in WORD_COUNTS}
```

We can see these functions in action on our test input word in the following code snippet, which shows lists of possible candidate words based on edit distances from the input word.

```
# input word
word = 'fianlly'
# zero edit distance from input word
edits0(word)

{'fianlly'}
```

```
# returns null set since it is not a valid word
known(edits0(word))

set()

# one edit distance from input word
edits1(word)

{'afianlly',
 'aianlly',
 .
 .
'yianlly',
'zfianlly',
'zianlly'}

# get correct words from above set
known(edits1(word))

{'finally'}

# two edit distances from input word
edits2(word)

{'fchnlly',
 'fianjlys',
  .
  .
 'fiapgnlly',
 'finanlqly'}

# get correct words from above set
known(edits2(word))

{'faintly', 'finally', 'finely', 'frankly'}
```

This output shows a set of valid candidate words that could be potential replacements for the incorrect input word. We select our candidate words from the list by giving higher priority to words whose edit distances are the smallest from the input word. The following code snippet illustrates this.

```
candidates = (known(edits0(word)) or
              known(edits1(word)) or
              known(edits2(word)) or
              [word])
candidates
```

```
{'finally'}
```

In case there is a tie in the candidates, we resolve it by taking the highest occurring word from our vocabulary dictionary WORD_COUNTS using the max(candidates, key=WORD_COUNTS.get) function. Thus, we now define our function to correct words using this logic.

```
def correct(word):
    """
    Get the best correct spelling for the input word
    """
    # Priority is for edit distance 0, then 1, then 2
    # else defaults to the input word itself.
    candidates = (known(edits0(word)) or
                  known(edits1(word)) or
                  known(edits2(word)) or
                  [word])
    return max(candidates, key=WORD_COUNTS.get)
```

We can use the function on incorrect words directly to correct them, as illustrated in the following snippet.

```
correct('fianlly')
```

```
'finally'
```

```
correct('FIANLLY')
```

```
'FIANLLY'
```

We see that this function is case sensitive and fails to correct words that are not lowercase, hence we write the following functions to make this generic to the case of words and correct their spelling regardless. The logic here is to preserve the original case of the word, convert it to lowercase, correct its spelling, and finally convert it back to its original case using the case_of function.

```
def correct_match(match):
    """
    Spell-correct word in match,
    and preserve proper upper/lower/title case.
    """
    word = match.group()
    def case_of(text):
        """
        Return the case-function appropriate
        for text: upper, lower, title, or just str.:
            """
        return (str.upper if text.isupper() else
                str.lower if text.islower() else
                str.title if text.istitle() else
                str)
    return case_of(word)(correct(word.lower()))

def correct_text_generic(text):
    """
    Correct all the words within a text,
    returning the corrected text.
    """
    return re.sub('[a-zA-Z]+', correct_match, text)
```

We can now use the function to correct words irrespective of their case, as illustrated in the following snippet.

```
correct_text_generic('fianlly')

'finally'

correct_text_generic('FIANLLY')

'FINALLY'
```

Of course this method is not always completely accurate and there might be words that are not corrected if they do not occur in our vocabulary dictionary. Using more data would help in this case as long as we cover different words with correct spellings in our vocabulary. This same algorithm is available to be used out-of-the-box in the TextBlob library. This is depicted in the following snippet.

```
from textblob import Word
w = Word('fianlly')
w.correct()
```

```
'finally'
```

```
# check suggestions
w.spellcheck()
```

```
[('finally', 1.0)]
```

```
# another example
w = Word('flaot')
w.spellcheck()
```

```
[('flat', 0.85), ('float', 0.15)]
```

Besides this, there are several robust libraries available in Python, including PyEnchant based on the enchant library (http://pythonhosted.org/pyenchant/), autocorrect, which is available at https://github.com/phatpiglet/autocorrect/, and aspell-python, which is a Python wrapper around the popular GNU Aspell. With the advent of deep learning, sequential models like RNNs and LSTMs coupled with word embeddings often out-perform these traditional methods. I also recommend readers take a look at DeepSpell, which is available at https://github.com/MajorTal/DeepSpell. It leverages deep learning to build a spelling corrector. Feel free to check them out and use them for correcting word spellings!

Stemming

To understand the process of stemming, we need to understand what word stems represent. In Chapter 1, we talked about morphemes, which are the smallest independent unit in any natural language. Morphemes consist of units that are stems and affixes. Affixes are units like prefixes, suffixes, and so on, which are attached to word stems to change their meaning or create a new word altogether. Word stems are also often known as the base form of a word and we can create new words by attaching affixes to them. This process is known as *inflection*. The reverse of this is obtaining the base form of a word from its inflected form and this is known as *stemming*.

Consider the word "JUMP", you can add affixes to it and form several new words like "JUMPS", "JUMPED", and "JUMPING". In this case, the base word is "JUMP" and this is the word stem. If we were to carry out stemming on any of its three inflected forms, we would get the base form. This is depicted more clearly in Figure 3-2.

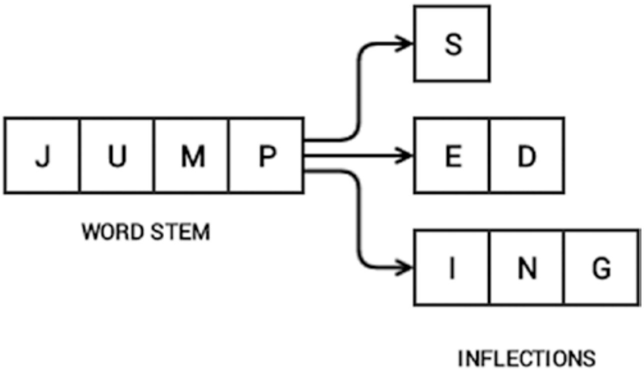


Figure 3-2. *Word stem and inflections*

Figure 3-2 depicts how the word stem is present in all its inflections since it forms the base on which each inflection is built upon using affixes. Stemming helps us standardize words to their base stem irrespective of their inflections, which helps many applications like classifying or clustering text or even in information retrieval. Search engines use such techniques extensively to give better accurate results irrespective of the word form. The NLTK package has several implementations for stemmers. These stemmers are implemented in the `stem` module, which inherits the `StemmerI` interface in the `nltk.stem.api` module. You can even create your own stemmer by using this class (technically it is an interface) as your base class. One of the most popular stemmers is the Porter stemmer, which is based on the algorithm developed by its inventor, Martin Porter. Originally, the algorithm is said to have a total of five different phases for reduction of inflections to their stems, where each phase has its own set of rules. There also exists a Porter2 algorithm, which was the original stemming algorithm with some improvements suggested by Dr. Martin Porter. You can see the Porter stemmer in action in the following code snippet.

```
# Porter Stemmer
In [458]: from nltk.stem import PorterStemmer
     ...: ps = PorterStemmer()
     ...: ps.stem('jumping'), ps.stem('jumps'), ps.stem('jumped')
(jump, jump, jump)

In [460]: ps.stem('lying')
'lie'

In [461]: ps.stem('strange')
'strang'
```

The Lancaster stemmer is based on the Lancaster stemming algorithm, also often known as the Paice/Husk stemmer, which was invented by Chris D. Paice. This stemmer is an iterative stemmer with over 120 rules, which specify specific removal or replacement for affixes to obtain the word stems. The following snippet shows the Lancaster stemmer in action.

```
# Lancaster Stemmer
In [465]: from nltk.stem import LancasterStemmer
     ...: ls = LancasterStemmer()
     ...:  print ls.stem('jumping'), ls.stem('jumps'), ls.stem('jumped')
(jump, jump, jump)

In [467]: ls.stem('lying')
'lying'

In [468]: ls.stem('strange')
'strange'
```

You can see the behavior of this stemmer is different from the previous Porter stemmer. Besides these two, there are several other stemmers, including `RegexpStemmer`, where you can build your own stemmer based on user-defined rules and `SnowballStemmer`, which supports stemming in 13 different languages besides English. The following code snippet shows some ways of using them for performing stemming. The `RegexpStemmer` uses regular expressions to identify the morphological affixes in words and any part of the string matching them is removed.

```
# Regex based stemmer
In [471]: from nltk.stem import RegexpStemmer
     ...: rs = RegexpStemmer('ing$|s$|ed$', min=4)
     ...:  rs.stem('jumping'), rs.stem('jumps'), rs.stem('jumped')
(jump, jump, jump)

In [473]: rs.stem('lying')
'ly'

In [474]: rs.stem('strange')
'strange'
```

You can see how the stemming results are different from the previous stemmers and is based completely on our custom defined rules based on regular expressions. The following snippet shows how we can use the SnowballStemmer to stem words in other languages. You can find more details about the Snowball Project at http://snowballstem.org/.

```
# Snowball Stemmer
In [486]: from nltk.stem import SnowballStemmer
     ...: ss = SnowballStemmer("german")
     ...: print('Supported Languages:', SnowballStemmer.languages)
Supported Languages: (u'danish', u'dutch', u'english', u'finnish',
u'french', u'german', u'hungarian', u'italian', u'norwegian', u'porter',
u'portuguese', u'romanian', u'russian', u'spanish', u'swedish')

# stemming on German words
# autobahnen -> cars
# autobahn -> car
In [488]: ss.stem('autobahnen')
'autobahn'

# springen -> jumping
# spring -> jump
In [489]: ss.stem('springen')
'spring'
```

The Porter stemmer is used most frequently, but you should choose your stemmer based on your problem and after trial and error. The following is a basic function that can be used for stemming text.

```
def simple_stemmer(text):
    ps = nltk.porter.PorterStemmer()
    text = ' '.join([ps.stem(word) for word in text.split()])
    return text

simple_stemmer("My system keeps crashing his crashed yesterday, ours 
crashes daily")

'My system keep crash hi crash yesterday, our crash daili'
```

Feel free to leverage this function for your own stemming needs. Also, if needed, you can even build your own stemmer with your own defined rules!

Lemmatization

The process of lemmatization is very similar to stemming, where we remove word affixes to get to a base form of the word. However in this case, this base form is also known as the *root* word but not the root stem. The difference between the two is that the root stem may not always be a lexicographically correct word, i.e., it may not be present in the dictionary but the root word, also known as the lemma, will always be present in the dictionary.

The lemmatization process is considerably slower than stemming because an additional step is involved where the root form or lemma is formed by removing the affix from the word if and only if the lemma is present in the dictionary. The NLTK package has a robust lemmatization module where it uses WordNet and the word's syntax and semantics like part of speech and context to get the root word or lemma. Remember from Chapter 1 when we discussed parts of speech? There were three entities of nouns, verbs, and adjectives that occur most frequently in natural language. The following code snippet depicts how to use lemmatization for words belonging to each of those types.

```
In [514]: from nltk.stem import WordNetLemmatizer
     ...: wnl = WordNetLemmatizer()

# lemmatize nouns
In [515]: print(wnl.lemmatize('cars', 'n'))
     ...: print(wnl.lemmatize('men', 'n'))
car
men

# lemmatize verbs
In [516]: print(wnl.lemmatize('running', 'v'))
     ...: print(wnl.lemmatize('ate', 'v'))
run
eat

# lemmatize adjectives
In [517]: print(wnl.lemmatize('saddest', 'a'))
     ...: print(wnl.lemmatize('fancier', 'a'))
sad
fancy
```

This snippet shows us how each word is converted to its base form using lemmatization. This helps us standardize words. This code leverages the `WordNetLemmatizer` class, which internally uses the `morphy()` function belonging to the `WordNetCorpusReader` class. This function basically finds the base form or lemma for a given word using the word and its part of speech by checking the WordNet corpus and uses a recursive technique for removing affixes from the word until a match is found in WordNet. If no match is found, the input word is returned unchanged. The part of speech is extremely important because if that is wrong, the lemmatization will not be effective, as you can see in the following snippet.

```
# ineffective lemmatization
In [518]: print wnl.lemmatize('ate', 'n')
     ...: print wnl.lemmatize('fancier', 'v')
ate
fancier
```

SpaCy makes things a lot easier since it performs parts of speech tagging and effective lemmatization for each token in a text document without you worrying about if you are using lemmatization effectively. The following function can be leveraged for performing effective lemmatization, thanks to spaCy!

```
import spacy
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)
text = 'My system keeps crashing his crashed yesterday, ours crashes daily'

def lemmatize_text(text):
    text = nlp(text)
    text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text
    for word in text])
    return text

lemmatize_text("My system keeps crashing! his crashed yesterday, ours
crashes daily")

'My system keep crash ! his crash yesterday , ours crash daily'
```

You can leverage NLTK or spaCy to build your own lemmatizers. Feel free to experiment with these functions on your own data.

Removing Stopwords

Stopwords are words that have little or no significance and are usually removed from text when processing it so as to retain words having maximum significance and context. Stopwords usually occur most frequently if you aggregate a corpus of text based on singular tokens and checked their frequencies. Words like “a,” “the,” “and,” and so on are stopwords. There is no universal or exhaustive list of stopwords and often each domain or language has its own set of stopwords. We depict a method to filter out and remove stopwords for English in the following code snippet.

```
from nltk.tokenize.toktok import ToktokTokenizer
tokenizer = ToktokTokenizer()
stopword_list = nltk.corpus.stopwords.words('english')
def remove_stopwords(text, is_lower_case=False):
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    if is_lower_case:
        filtered_tokens = [token for token in tokens if token not in
        stopword_list]
    else:
        filtered_tokens = [token for token in tokens if token.lower() not
        in stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

```
remove_stopwords("The, and, if are stopwords, computer is not")
```

```
', , stopwords , computer'
```

There is no universal stopword list, but we use a standard English language stopwords list from NLTK. You can also add your own domain-specific stopwords as needed. In the previous function, we leverage the use of NLTK, which has a list of stopwords for English, and use it to filter out all tokens that correspond to stopwords. This output shows us a reduced number of tokens compared to what we had earlier and you can compare and check the tokens that were removed as stopwords. To see the list of all English stopwords in NLTK's vocabulary, you can print the contents of `nltk.corpus.stopwords.words('english')` to get an idea of the various stopwords. One important thing to remember is that negations like "not" and "no" are removed in this case (in the first sentence) and often it is essential to preserve them so as the actual meaning of the sentence is not lost in applications like sentiment analysis. So you would need to make sure you do not remove these words in those scenarios.

Bringing It All Together—Building a Text Normalizer

Let's now bring everything we learned together and chain these operations to build a text normalizer to preprocess text data. We focus on including the major components often used for text wrangling in our custom function.

```
def normalize_corpus(corpus, html_stripping=True, contraction_expansion=True,
                     accented_char_removal=True, text_lower_case=True,
                     text_lemmatization=True, special_char_removal=True,
                     stopword_removal=True, remove_digits=True):
```

```
    normalized_corpus = []
    # normalize each document in the corpus
    for doc in corpus:
        # strip HTML
        if html_stripping:
            doc = strip_html_tags(doc)
        # remove accented characters
        if accented_char_removal:
            doc = remove_accented_chars(doc)
        # expand contractions
        if contraction_expansion:
            doc = expand_contractions(doc)
        # lowercase the text
        if text_lower_case:
            doc = doc.lower()
        # remove extra newlines
        doc = re.sub(r'[\r|\n|\r\n]+', ' ',doc)
        # lemmatize text
        if text_lemmatization:
            doc = lemmatize_text(doc)
        # remove special characters and\or digits
        if special_char_removal:
            # insert spaces between special characters to isolate them
            special_char_pattern = re.compile(r'([{.(-)!}])')
            doc = special_char_pattern.sub(" \\1 ", doc)
            doc = remove_special_characters(doc, remove_digits=remove_digits)
        # remove extra whitespace
        doc = re.sub(' +', ' ', doc)
        # remove stopwords
        if stopword_removal:
            doc = remove_stopwords(doc, is_lower_case=text_lower_case)

        normalized_corpus.append(doc)

    return normalized_corpus
```

Let's now put this function in action! We will leverage our sample text from the previous sections as the input document, which we will preprocess using the preceding function.

```
{'Original': sample_text,
 'Processed': normalize_corpus([sample_text])[0]}
```

```
{'Original': "US unveils world's most powerful supercomputer, beats
China. The US has unveiled the world's most powerful supercomputer called
'Summit', beating the previous record-holder China's Sunway TaihuLight.
With a peak performance of 200,000 trillion calculations per second,
it is over twice as fast as Sunway TaihuLight, which is capable of
93,000 trillion calculations per second. Summit has 4,608 servers, which
reportedly take up the size of two tennis courts.",
```

```
 'Processed': 'us unveil world powerful supercomputer beat china us unveil
world powerful supercomputer call summit beat previous record holder chinas
sunway taihulight peak performance trillion calculation per second twice
fast sunway taihulight capable trillion calculation per second summit
server reportedly take size two tennis court'}
```

Thus, you can see how our text preprocessor helps in preprocessing our sample news article! In the next section, we look at ways of analyzing and understanding various facets of textual data with regard to its syntactic properties and structure.

Understanding Text Syntax and Structure

We talked about language syntax and structure in detail in Chapter 1. If you don't remember the basics, head over to the section titled "Language Syntax and Structure" in Chapter 1 and skim through it quickly to get an idea of the various ways of analyzing and understanding the syntax and structure of textual data. To refresh your memory, let's briefly cover the importance of text syntax and structure.

For any language, syntax and structure usually go hand in hand, where a set of specific rules, conventions, and principles govern the way words are combined into phrases; phrases are combined into clauses; and clauses are combined into sentences. We will be talking specifically about the English language syntax and structure in this section. In English, words usually combine to form other constituent units. These constituents include words, phrases, clauses, and sentences. The sentence "The brown fox is quick and he is jumping over the lazy dog" is made of a bunch of words. Just looking at the words by themselves doesn't tell us much (see Figure 3-3).

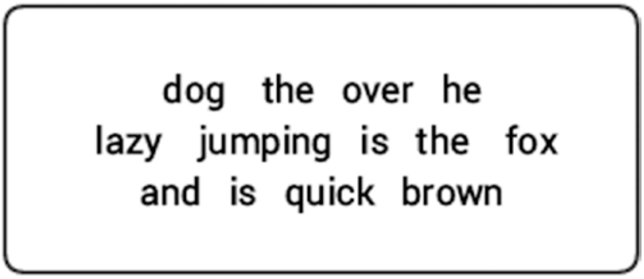


Figure 3-3. *A bunch of unordered words doesn't convey much information*

Knowledge about the structure and syntax of language is helpful in many areas like text processing, annotation, and parsing for further operations such as text classification or summarization. In this section, we implement some of the concepts and techniques used to understand text syntax and structure. This is extremely useful in natural language processing and is usually done after text processing and wrangling. We focus on implementing the following techniques:

- Parts of speech (POS) tagging
- Shallow parsing or chunking
- Dependency parsing
- Constituency parsing

This book is targeted toward practitioners and enforces and emphasizes on best approaches for implementing and using techniques and algorithms in real-world problems. Hence in the following sections, we look at the best possible ways of leveraging libraries like NLTK and spaCy to implement some of these techniques. Besides this, since you might be interested in the internals and implementing some of these techniques on your own, we also look at ways to accomplish this. Before jumping into the details, we look at the necessary dependencies and installation details for the required libraries, since some of them are not very straightforward.

Installing Necessary Dependencies

We leverage several libraries and dependencies:

- The nltk library
- The spacy library
- The Stanford Parser
- Graphviz and necessary libraries for visualization

We touched upon installing NLTK in Chapter 1. You can install it directly by going to your terminal or command prompt and typing pip install nltk, which will download and install it. Remember to install the library preferably equal to or higher than version 3.2.4. After downloading and installing NLTK, remember to download the corpora, which we also discussed in Chapter 1. For more details on downloading and installing NLTK, you can follow the information at http://www.nltk.org/install.html and http://www.nltk.org/data.html, which tells you how to install the data dependencies. Start the Python interpreter and use the following snippet.

```
import nltk
# download all dependencies and corpora
nltk.download('all', halt_on_error=False)
# OR use a GUI based downloader and select dependencies
nltk.download()
```

To install spaCy, type pip install spacy from the terminal or conda install spaCy. Once it's done, download the English language model using the command, python -m spacy.en.download from the terminal, which will download around 500MB of data in the directory of the spaCy package. For more details, you can refer to the link https://spacy.io/docs/#getting-started, which tells you how to get started with using spaCy. We will use spaCy for tagging and depicting dependency based parsing. However, in case you face issues loading spaCy's language models, feel free to follow the steps highlighted here to resolve this issue. (I faced this issue in one of my systems but it doesn't always occur.)

```
# OPTIONAL: ONLY USE IF SPACY FAILS TO LOAD LANGUAGE MODEL
# Use the following command to install spaCy
> pip install -U spacy
OR
> conda install -c conda-forge spacy

# Download the following language model and store it in disk
https://github.com/explosion/spacy-models/releases/tag/en_core_web_md-2.0.0

# Link the same to spacy
> python -m spacy link ./spacymodels/en_core_web_md-2.0.0/en_core_web_md
en_core Linking successful
    ./spacymodels/en_core_web_md-2.0.0/en_core_web_md --> ./Anaconda3/lib/
site-packages/spacy/data/en_core
You can now load the model via spacy.load('en_core')
```

The Stanford Parser is a Java-based implementation for a language parser developed at Stanford, which helps parse sentences to understand their underlying structure. We perform both dependency and constituency grammar based parsing using the Stanford Parser and NLTK, which provides an excellent wrapper to leverage and use the parser from Python itself without the need to write code in Java. You can refer to the official installation guide at https://github.com/nltk/nltk/wiki/Installing-Third-Party-Software, which tells us how to download and install the Stanford Parser and integrate it with NLTK. Personally, I faced several issues especially in Windows based systems; hence, I will provide one of the best known methods for installation of the Stanford Parser and its necessary dependencies.

To start with, make sure you download and install the Java Development Kit (not just JRE also known as Java Runtime Environment) by going to http://www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=otnjp. Use any version typically on or after Java SE 8u101 / 8u102. I used 8u102 since I haven't upgraded Java in a while. After installing, make sure that you have set the path for Java by adding it to the path system environment variable. You can also create a JAVA_HOME environment variable pointing to the java.exe file belonging to the JDK.

In my experience neither worked for me when running the code from Python and I had to explicitly use the Python os library to set the environment variable, which I will be showing when we dive into the implementation details. Once Java is installed, download the official Stanford Parser from http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip, which seems to work quite well. You can try a later version by going to http://nlp.stanford.edu/software/lex-parser.shtml#Download and checking the "Release History" section. After downloading, unzip it to a known location in your filesystem. Once you're done, you are now ready to use the parser from NLTK, which we will be exploring soon!

Graphviz is not a necessity and we will only be using it to view the dependency parse tree generated by the Stanford Parser. You can download Graphviz from its official website at http://www.graphviz.org/Download_windows.php and install it. Next you need to install pygraphviz, which you can get by downloading the wheel file from http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygraphviz based on your system architecture and Python version. Then install it using the pip install pygraphviz-1.3.1-cp34-none-win_amd64.whl command for a 64-bit system running Python 3.x. Once it's installed, pygraphviz should be ready to work. Some people reported running into additional issues and you might need to install pydot-ng and graphviz in the same order using the following snippet in the terminal.

```
pip install pydot-ng
pip install graphviz
```

We also leverage NLTK's plotting capabilities to visualize parse trees in Jupyter notebooks. To enable this, you might need to install ghostscript in case NLTK throws an error. Instructions for installation and setup are depicted as follows.

```
## download and install ghostscript from https://www.ghostscript.com/
download/gsdnld.html

# often need to add to the path manually (for windows)
os.environ['PATH'] = os.environ['PATH']+";C:\\Program Files\\gs\\gs9.09\\bin\\"
```

With this, we are done with installing our necessary dependencies and can start implementing and looking at practical examples. However, we are not ready just yet. We need to go through a few basic concepts of machine learning before we dive into the code and examples.

Important Machine Learning Concepts

We will be implementing and training some of our own taggers in the following section using corpora and leverage existing taggers. There are some important concepts related to analytics and machine learning, which you must know to understand the implementations more clearly.

- **Data preparation:** Usually consists of preprocessing the data before extracting features and training
- **Feature extraction:** The process of extracting useful features from raw data that are used to train machine learning models
- **Features:** Various useful attributes of the data (examples could be age, weight, and so on for personal data)
- **Training data:** A set of data points used to train a model
- **Testing/validation data:** A set of data points on which a pretrained model is tested and evaluated to see how well it performs
- **Model:** This is built using a combination of data/features and a machine learning algorithm that could be supervised or unsupervised
- **Accuracy:** How well the model predicts something (also has other detailed evaluation metrics like precision, recall, and F1-score)

These terms should be enough to get you started. Going into details is beyond the current scope; however, you will find a lot of resources on the web on machine learning if you are interested in exploring some of them further. We recommend checking out *Practical Machine Learning with Python*, Apress 2018, if you are interested in learning machine learning using a hands-on approach. Besides this, we cover supervised and unsupervised learning with regards to textual data in subsequent chapters.

Parts of Speech Tagging

Parts of speech (POS) are specific lexical categories to which words are assigned based on their syntactic context and role. If you remember from Chapter 1, we covered some ground on POS, where we mentioned the main POS being nouns, verbs, adjectives, and adverbs. The process of classifying and labeling POS tags for words is defined as parts of speech tagging (POS tagging).

POS tags are used to annotate words and depict their POS, which is really helpful when we need to use the same annotated text later in NLP-based applications because we can filter by specific parts of speech an-d utilize that information to perform specific analysis. We can narrow down nouns and determine which ones are the most prominent. Considering our previous example sentence, "The brown fox is quick and he is jumping over the lazy dog", if we were to annotate it using basic POS tags, it would look like Figure 3-4.

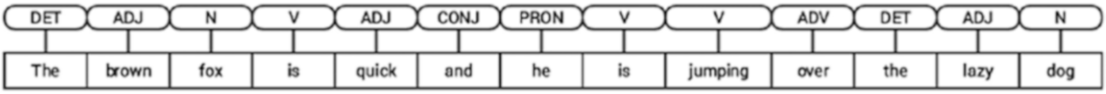


***Figure 3-4.** POS tagging for a sentence*

Thus, a sentence typically follows a hierarchical structure consisting of the following components: sentence → clauses → phrases → words.

We will be using the Penn Treebank notation for POS tagging and most of the recommended POS taggers also leverage it. You can find out more information about various POS tags and their notation at `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf`, which contains detailed documentation explaining each tag with examples. The Penn Treebank project is a part of the University of Pennsylvania and their web page can be found at `https://catalog.ldc.upenn.edu/docs/LDC95T7/treebank2.index.html`, which gives more information about the project. Remember there are various tags, such as POS tags for parts of speech assigned to words, chunk tags, which are usually assigned to phrases, and some tags are secondary tags, which are used to depict relations.

Table 3-1 provides a detailed overview of different tags with examples in case you do not want to go through the detailed documentation for Penn Treebank tags. You can use this as a reference anytime to understand POS tags and parse trees in a better way.

***Table 3-1.** Parts of Speech Tags*

SI No.	TAG	DESCRIPTION	EXAMPLE(S)
1	CC	Coordinating conjunction	and, or
2	CD	Cardinal number	five, one, 2
3	DT	Determiner	a, the
4	EX	Existential *there*	there were two cars
5	FW	Foreign word	d'hoevre, mais
6	IN	Preposition/subordinating conjunction	of, in, on, that
7	JJ	Adjective	quick, lazy
8	JJR	Adjective, comparative	quicker, lazier
9	JJS	Adjective, superlative	quickest, laziest
10	LS	List item marker	2)
11	MD	Verb, modal	could, should
12	NN	Noun, singular or mass	fox, dog
13	NNS	Noun, plural	foxes, dogs
14	NNP	Noun, proper singular	John, Alice
15	NNPS	Noun, proper plural	Vikings, Indians, Germans
16	PDT	Predeterminer	both cats
17	POS	Possessive ending	boss's
18	PRP	Pronoun, personal	me, you
19	PRP$	Pronoun, possessive	our, my, your
20	RB	Adverb	naturally, extremely, hardly
21	RBR	Adverb, comparative	better
22	RBS	Adverb, superlative	best
23	RP	Adverb, particle	about, up
24	SYM	Symbol	%, $
25	TO	Infinitival to	how to, what to do

(*continued*)

Table 3-1. (*continued*)

SI No.	TAG	DESCRIPTION	EXAMPLE(S)
26	UH	Interjection	oh, gosh, wow
27	VB	Verb, base form	run, give
28	VBD	Verb, past tense	ran, gave
29	VBG	Verb, gerund/present participle	running, giving
30	VBN	Verb, past participle	given
31	VBP	Verb, non-third person singular present	I think, I take
32	VBZ	Verb, third person singular present	he thinks, he takes
33	WDT	Wh-determiner	which, whatever
34	WP	Wh-pronoun, personal	who, what
35	WP$	Wh-pronoun, possessive	whose
36	WRB	Wh-adverb	where, when
37	NP	Noun phrase	the brown fox
38	PP	Prepositional phrase	in between, over the dog
39	VP	Verb phrase	was jumping
40	ADJP	Adjective phrase	warm and snug
41	ADVP	Adverb phrase	also
42	SBAR	Subordinating conjunction	whether or not
43	PRT	Particle	up
44	INTJ	Interjection	hello
45	PNP	Prepositional noun phrase	over the dog, as of today
46	-SBJ	Sentence subject	the fox jumped over the dog
47	-OBJ	Sentence object	the fox jumped over the dog

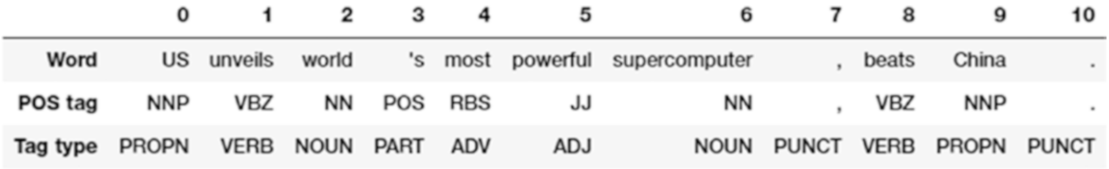
This table shows us the main POS tag set used in the Penn Treebank and is the most widely used POS tag set in various text analytics and NLP applications. In the following sections, we look at some hands-on implementations of POS tagging.

Building POS Taggers

We will be leveraging NLTK and spaCy, which use the *Penn Treebank notation* for POS tagging. To demonstrate how things work, we will leverage a news headline from our sample news article from the previous sections. Let's look at how POS tagging can be implemented using spaCy. See Figure 3-5.

```
sentence = "US unveils world's most powerful supercomputer, beats China."
```

```
import pandas as pd
import spacy
nlp = spacy.load('en_core', parse=True, tag=True, entity=True)
sentence_nlp = nlp(sentence)
# POS tagging with Spacy
spacy_pos_tagged = [(word, word.tag_, word.pos_) for word in sentence_nlp]
pd.DataFrame(spacy_pos_tagged, columns=['Word', 'POS tag', 'Tag type']).T
```

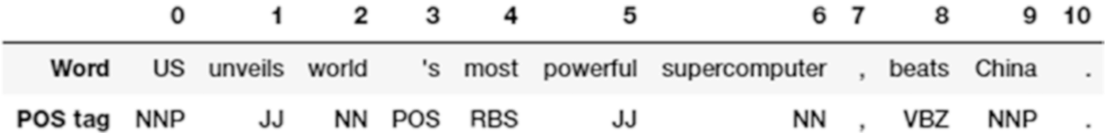


	0	1	2	3	4	5	6	7	8	9	10
Word	US	unveils	world	's	most	powerful	supercomputer	,	beats	China	.
POS tag	NNP	VBZ	NN	POS	RBS	JJ	NN	,	VBZ	NNP	.
Tag type	PROPN	VERB	NOUN	PART	ADV	ADJ	NOUN	PUNCT	VERB	PROPN	PUNCT

Figure 3-5. *POS tagging for our news headline using spaCy*

Thus, we can clearly see in Figure 3-5 the POS tag for each token in our sample news headline, as defined using spaCy, and they make perfect sense. Let's try to perform the same task using NLTK (see Figure 3-6).

```
# POS tagging with nltk
import nltk
nltk_pos_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
pd.DataFrame(nltk_pos_tagged, columns=['Word', 'POS tag']).T
```



	0	1	2	3	4	5	6	7	8	9	10
Word	US	unveils	world	's	most	powerful	supercomputer	,	beats	China	.
POS tag	NNP	JJ	NN	POS	RBS	JJ	NN	,	VBZ	NNP	.

Figure 3-6. *POS tagging for our news headline using NLTK*

The output in Figure 3-6 gives us tags that purely follow the Penn Treebank format specifying the specific form of adjective, noun, or verbs in more detail.

We will now explore some techniques to build our own POS taggers! We leverage some classes provided by NLTK. To evaluate the performance of our taggers, we use some test data from the treebank corpus in NLTK. We will also be using some training data for training some of our taggers. To start with, we will first get the necessary data for training and evaluating the taggers by reading in the tagged treebank corpus.

```
from nltk.corpus import treebank
data = treebank.tagged_sents()
train_data = data[:3500]
test_data = data[3500:]
print(train_data[0])
```

```
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years',
'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the',
'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'),
('director', 'NN'), ('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

We will use the test data to evaluate our taggers and see how they work on our sample sentence by using its tokens as input. All the taggers we will be leveraging from NLTK are a part of the nltk.tag package. Each tagger is a child class of the base TaggerI class and each tagger implements a tag() function, which takes a list of sentence tokens as input and returns the same list of words with their POS tags as output. Besides tagging, there is also an evaluate() function, which is used to evaluate the performance of the tagger. This is done by tagging each input test sentence and then comparing the result with the actual tags of the sentence. We will be using the same function to test the performance of our taggers on test_data.

We will first look at the DefaultTagger, which inherits from the SequentialBackoffTagger base class and assigns the same user input POS tag to each word. This might seem to be really naïve but it is an excellent way to form a baseline POS tagger and improve upon it.

```
# default tagger
from nltk.tag import DefaultTagger
dt = DefaultTagger('NN')
```

```
# accuracy on test data
dt.evaluate(test_data)
```

```
0.1454158195372253
```

```
# tagging our sample headline
dt.tag(nltk.word_tokenize(sentence))
```

```
[('US', 'NN'), ('unveils', 'NN'), ('world', 'NN'), ("'s", 'NN'), ('most', 'NN'),
 ('powerful', 'NN'), ('supercomputer', 'NN'), (',', 'NN'), ('beats', 'NN'),
 ('China', 'NN'), ('.', 'NN')]
```

We can see from this output we have obtained 14% accuracy in correctly tagging words from the treebank test dataset, which is not great. The output tags on our sample sentence are all nouns, just like we expected since we fed the tagger with the same tag. We will now use regular expressions and the RegexpTagger to see if we can build a better performing tagger.

```
# regex tagger
from nltk.tag import RegexpTagger
# define regex tag patterns
patterns = [
        (r'.*ing$', 'VBG'),               # gerunds
        (r'.*ed$', 'VBD'),                # simple past
        (r'.*es$', 'VBZ'),                # 3rd singular present
        (r'.*ould$', 'MD'),               # modals
        (r'.*\'s$', 'NN$'),               # possessive nouns
        (r'.*s$', 'NNS'),                 # plural nouns
        (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
        (r'.*', 'NN')                     # nouns (default) ...
]
rt = RegexpTagger(patterns)
```

```
# accuracy on test data
rt.evaluate(test_data)
```

```
0.24039113176493368
```

```
# tagging our sample headline
```

```
rt.tag(nltk.word_tokenize(sentence))
```

```
[('US', 'NN'), ('unveils', 'NNS'), ('world', 'NN'), ("'s", 'NN$'), ('most', 'NN'),
 ('powerful', 'NN'), ('supercomputer', 'NN'), (',', 'NN'), ('beats', 'NNS'),
 ('China', 'NN'), ('.', 'NN')]
```

This output shows us that the accuracy has now increased to 24%, but can we do better? We will now train some n-gram taggers. If you don't know already, n-grams are contiguous sequences of **n** items from a sequence of text or speech. These items could consist of words, phonemes, letters, characters, or syllables. Shingles are n-grams where the items only consist of words. We will use n-grams of size 1, 2, and 3, which are also known as unigram, bigram, and trigram, respectively. The `UnigramTagger`, `BigramTagger`, and `TrigramTagger` are classes that inherit from the base class `NGramTagger`, which itself inherits from the `ContextTagger` class, which inherits from the `SequentialBackoffTagger` class. We will use the `train_data` as training data to train the n-gram taggers based on sentence tokens and their POS tags. Then we will evaluate the trained taggers on `test_data` and see the result upon tagging our sample sentence.

```
## N gram taggers
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

ut = UnigramTagger(train_data)
bt = BigramTagger(train_data)
tt = TrigramTagger(train_data)

# testing performance of unigram tagger
print(ut.evaluate(test_data))
print(ut.tag(nltk.word_tokenize(sentence)))
```

```
0.8619421047536063
[('US', 'NNP'), ('unveils', None), ('world', 'NN'), ("'s", 'POS'), ('most',
'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','), ('beats',
None), ('China', 'NNP'), ('.', '.')]
```

```
# testing performance of bigram tagger
print(bt.evaluate(test_data))
print(bt.tag(nltk.word_tokenize(sentence)))
```

```
0.13592796979378 45
[('US', None), ('unveils', None), ('world', None), ("'s", None), ('most',
None), ('powerful', None), ('supercomputer', None), (',', None), ('beats',
None), ('China', None), ('.', None)]
```

```
# testing performance of trigram tagger
print(tt.evaluate(test_data))
print(tt.tag(nltk.word_tokenize(sentence)))
```

```
0.08142124116565011
[('US', None), ('unveils', None), ('world', None), ("'s", None), ('most',
None), ('powerful', None), ('supercomputer', None), (',', None), ('beats',
None), ('China', None), ('.', None)]
```

This output clearly shows us that we obtain 86% accuracy on the test set using unigram tagger alone, which is really good compared to our last tagger. The None tag indicates the tagger was unable to tag that word and the reason for that would be that it was unable to get a similar token in the training data. Accuracies of the bigram and trigram models are far lower because the same bigrams and trigrams observed in the training data aren't always present in the same way in the testing data.

We now look at an approach to combine all the taggers by creating a combined tagger with a list of taggers and use a backoff tagger. Essentially, we would create a chain of taggers and each tagger would fall back on a backoff tagger if it cannot tag the input tokens.

```
def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff

ct = combined_tagger(train_data=train_data,
                     taggers=[UnigramTagger, BigramTagger, TrigramTagger],
                     backoff=rt)

# evaluating the new combined tagger with backoff taggers
print(ct.evaluate(test_data))
print(ct.tag(nltk.word_tokenize(sentence)))
```

```
0.9108335753703166
[('US', 'NNP'), ('unveils', 'NNS'), ('world', 'NN'), ("'s", 'POS'),
('most', 'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','),
('beats', 'NNS'), ('China', 'NNP'), ('.', '.')]
```

We now obtain an accuracy of 91% on the test data, which is excellent. Also we see that this new tagger can successfully tag all the tokens in our sample sentence (even though a couple of them are not correct, like beats should be a verb).

For our final tagger, we will use a supervised classification algorithm to train our tagger. The ClassifierBasedPOSTagger class enables us train a tagger by using a supervised learning algorithm in the classifier_builder parameter. This class is inherited from the ClassifierBasedTagger and it has a feature_detector() function that forms the core of the training process. This function is used to generate various features from the training data like word, previous word, tag, previous tag, case, and so on. In fact, you can even build your own feature detector function and pass it to the feature_detector parameter when instantiating an object of the ClassifierBasedPOSTagger class.

The classifier we will be using is the NaiveBayesClassifier. It uses the Bayes' theorem to build a probabilistic classifier assuming the features are independent. You can read more about it at https://en.wikipedia.org/wiki/Naive_Bayes_classifier since going into details about the algorithm is out of our current scope. The following code snippet shows a classification based approach to building and evaluating a POS tagger.

```
from nltk.classify import NaiveBayesClassifier, MaxentClassifier
from nltk.tag.sequential import ClassifierBasedPOSTagger

nbt = ClassifierBasedPOSTagger(train=train_data,
                              classifier_builder=NaiveBayesClassifier.train)

# evaluate tagger on test data and sample sentence
print(nbt.evaluate(test_data))
print(nbt.tag(nltk.word_tokenize(sentence)))
```

```
0.9306806079969019
[('US', 'PRP'), ('unveils', 'VBZ'), ('world', 'VBN'), ("'s", 'POS'),
('most', 'JJS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','),
('beats', 'VBZ'), ('China', 'NNP'), ('.', '.')]
```

Using this tagger, we get an accuracy of 93% on our test data, which is the highest out of all our taggers. Also if you observe the output tags for the sample sentence, you will see they are correct and make perfect sense. This gives us an idea of how powerful and effective classifier based POS taggers can be! Feel free to use a different classifier like `MaxentClassifier` and compare the performance with this tagger. We have included the code in the notebook to make things easier.

There are also several other ways to build and use POS taggers using NLTK and other packages. Even though it is not necessary and this should be enough to cover your POS tagging needs, you can go ahead and explore other methods to compare with these methods and satisfy your curiosity.

Shallow Parsing or Chunking

Shallow parsing, also known as *light parsing* or *chunking,* is a technique of analyzing the structure of a sentence to break it down into its smallest constituents, which are tokens like words, and group them together into higher-level phrases. In shallow parsing, there is more focus on identifying these phrases or chunks rather than diving into further details of the internal syntax and relations inside each chunk, like we see in grammar based parse trees obtained from deep parsing. The main objective of shallow parsing is to obtain semantically meaningful phrases and observe relations among them.

You can look at the "Language Syntax and Structure" section from Chapter 1 just to refresh your memory regarding how words and phrases give structure to a sentence consisting of a bunch of words. Based on the hierarchy we depicted earlier, groups of words make up phrases. There are five major categories of phrases:

- **Noun phrase (NP):** These are phrases where a noun acts as the head word. Noun phrases act as a subject or object to a verb.
- **Verb phrase (VP):** These phrases are lexical units that have a verb acting as the head word. Usually, there are two forms of verb phrases. One form has the verb components as well as other entities such as nouns, adjectives, or adverbs as parts of the object.
- **Adjective phrase (ADJP):** These are phrases with an adjective as the head word. Their main role is to describe or qualify nouns and pronouns in a sentence, and they will be placed before or after the noun or pronoun.

- **Adverb phrase (ADVP):** These phrases act like adverbs since the adverb acts as the head word in the phrase. Adverb phrases are used as modifiers for nouns, verbs, or adverbs by providing further details that describe or qualify them.
- **Prepositional phrase (PP):** These phrases usually contain a preposition as the head word and other lexical components like nouns, pronouns, and so on. These act like an adjective or adverb, describing other words or phrases.

A shallow parsed tree is depicted in Figure 3-7 for a sample sentence just to refresh your memory on its structure.

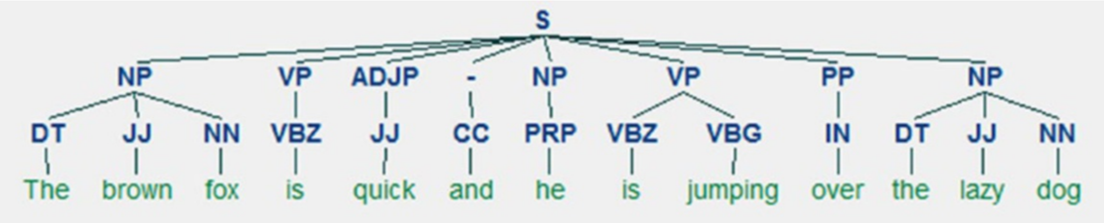


***Figure 3-7.** An example of shallow parsing depicting higher level phrase annotations*

We will now look at ways in which we can implement shallow parsing on text data using a wide variety of techniques, including regular expressions, chunking, chinking, and tag based training.

Building Shallow Parsers

We use several techniques like regular expressions and tagging based learners to build our own shallow parsers. Just like POS tagging, we use some training data to train our parsers if needed and evaluate all our parsers on some test data and on our sample sentence. The treebank corpus is available in NLTK with chunk annotations. We load it and then prepare our training and testing datasets using the following code snippet.

```
from nltk.corpus import treebank_chunk
data = treebank_chunk.chunked_sents()

train_data = data[:3500]
test_data = data[3500:]
```

```
# view sample data
print(train_data[7])
```

```
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

From this output, you can see that our data points are sentences and are already annotated with phrase and POS tags metadata, which will be useful in training shallow parsers. We start by using regular expressions for shallow parsing using concepts of chunking and chinking. Using the process of *chunking*, we can use and specify specific patterns to identify what we would want to chunk or segment in a sentence, such as phrases based on specific metadata. *Chinking* is the reverse of chunking, where we specify which specific tokens we do not want to be a part of any chunk and then form the necessary chunks excluding these tokens. Let's consider a simple sentence (our news headline) and use regular expressions. We leverage the RegexpParser class to create shallow parsers to illustrate chunking and chinking for noun phrases.

```
from nltk.chunk import RegexpParser

# get POS tagged sentence
tagged_simple_sent = nltk.pos_tag(nltk.word_tokenize(sentence))
print('POS Tags:', tagged_simple_sent)

# illustrate NP chunking based on explicit chunk patterns
chunk_grammar = """
NP: {<DT>?<JJ>*<NN.*>}
"""
rc = RegexpParser(chunk_grammar)
c = rc.parse(tagged_simple_sent)

# print and view chunked sentence using chunking
```

```
print(c)
c

(S
  (NP US/NNP)
  (NP unveils/JJ world/NN)
  's/POS
  most/RBS
  (NP powerful/JJ supercomputer/NN)
  ,/,
  beats/VBZ
  (NP China/NNP)
  ./.)
```

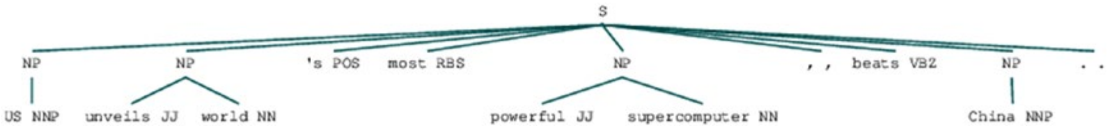


Figure 3-8. *Shallow parsing using chunking*

We can see how the shallow parse tree looks in Figure 3-8 with only NP chunks using chunking. Let's look at building this using chinking now.

```
# illustrate NP chunking based on explicit chink patterns
chink_grammar = """
NP:
    {<.*>+}             # Chunk everything as NP
    }<VBZ|VBD|JJ|IN>+{  # Chink sequences of VBD\VBZ\JJ\IN
"""
rc = RegexpParser(chink_grammar)
c = rc.parse(tagged_simple_sent)

# print and view chunked sentence using chinking
print(c)
c

(S
  (NP US/NNP)
  unveils/JJ
  (NP world/NN 's/POS most/RBS)
  powerful/JJ
  (NP supercomputer/NN ,/,)
  beats/VBZ
  (NP China/NNP ./.))
```

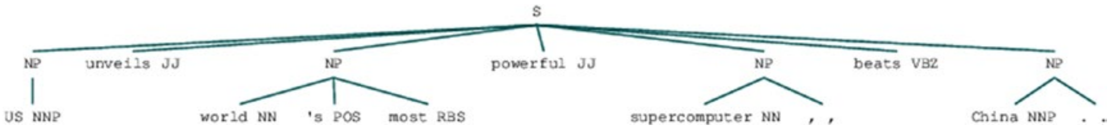


Figure 3-9. *Shallow parsing using chinking*

We can see how the shallow parse tree looks in Figure 3-9, with verbs and adjectives forming chinks and separating out noun phrases. Remember that *chunks* are sequences of tokens that are included in a collective group (chunk) and *chinks* are tokens or sequences of tokens that are excluded from chunks. We will now train a more generic regular expression-based shallow parser and test its performance on our test treebank data. Internally there are several steps that are executed to perform this parsing. The Tree structures used to represent parsed sentences in NLTK are converted to ChunkString objects. We create an object of RegexpParser using defined chunking and chinking rules. Objects of the ChunkRule and ChinkRule classes help create the complete shallow parsed tree with the necessary chunks based on specified patterns. The following code snippet represents a shallow parser using regular expression based patterns.

```
# create a more generic shallow parser
grammar = """
NP: {<DT>?<JJ>?<NN.*>}
ADJP: {<JJ>}
ADVP: {<RB.*>}
PP: {<IN>}
VP: {<MD>?<VB.*>+}
"""
rc = RegexpParser(grammar)
c = rc.parse(tagged_simple_sent)
```

```
# print and view shallow parsed sample sentence
print(c)
c
```

```
(S
  (NP US/NNP)
  (NP unveils/JJ world/NN)
  's/POS
  (ADVP most/RBS)
  (NP powerful/JJ supercomputer/NN)
  ,/,
  (VP beats/VBZ)
  (NP China/NNP)
  ./.)
```

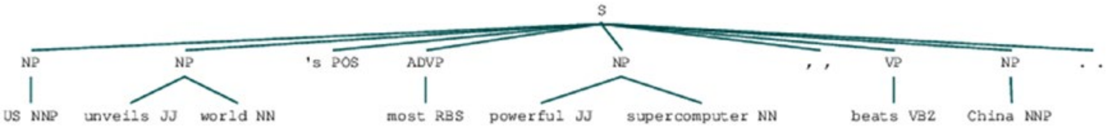


***Figure 3-10.** Shallow parsing using more specific rules*

We can see how the shallow parse tree looks in Figure 3-10, with more specific rules for specific phrases. Let's take a look at how this parser performs on the test dataset we built earlier.

```
# Evaluate parser performance on test data
print(rc.evaluate(test_data))
```

```
ChunkParse score:
    IOB Accuracy:  46.1%%
    Precision:     19.9%%
    Recall:        43.3%%
    F-Measure:     27.3%%
```

From the output, we can see that the parse tree for our sample sentence is very similar to the one we obtained from the out-of-the-box parser in the previous section. Also the accuracy of the overall test data is 54.5%, which is quite decent for a start. To get more details as to what each performance metric signifies, you can refer to the "Evaluating Classification Models" section in Chapter 5.

Remember when we said annotated tagged metadata for text is useful in many ways? We use the chunked and tagged `treebank` training data now to build a shallow parser. We leverage two chunking utility functions—`tree2conlltags` to get triples of word, tag, and chunk tags for each token and `conlltags2tree` to generate a parse tree from these token triples.

We use these functions to train our parser later. First, let's see how these two functions work. Remember the chunk tags use a popular format, known as the IOB format. In this format, you will notice some new notations with `I`, `O`, and `B` prefixes, which is the popular IOB notation used in chunking. It depicts Inside, Outside, and Beginning. The `B-` prefix before a tag indicates it is the beginning of a chunk; the `I-` prefix indicates that it is inside a chunk. The `O` tag indicates that the token does not belong to any chunk. The `B-` tag is always used when there are subsequent tags following it of the same type without the presence of `O` tags between them.

```
from nltk.chunk.util import tree2conlltags, conlltags2tree
# look at a sample training tagged sentence
train_sent = train_data[7]
print(train_sent)
```

```
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

```
# get the (word, POS tag, Chunk tag) triples for each token
wtc = tree2conlltags(train_sent)
wtc
```

```
[('A', 'DT', 'B-NP'),
 ('Lorillard', 'NNP', 'I-NP'),
 ('spokewoman', 'NN', 'I-NP'),
 ('said', 'VBD', 'O'),
 (',', ',', 'O'),
 ('``', '``', 'O'),
 ('This', 'DT', 'B-NP'),
 ('is', 'VBZ', 'O'),
 ('an', 'DT', 'B-NP'),
 ('old', 'JJ', 'I-NP'),
 ('story', 'NN', 'I-NP'),
 ('.', '.', 'O')]
```

```
# get shallow parsed tree back from the WTC triples
tree = conlltags2tree(wtc)
print(tree)
```

```
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

Now that we know how these functions work, we define a function called conll_tag_chunks() to extract POS and Chunk tags from sentences with chunked annotations and reuse our combined_taggers() function from POS tagging to train multiple taggers with backoff taggers, as depicted in the following code snippet.

```
def conll_tag_chunks(chunk_sents):
  tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]
  return [[(t, c) for (w, t, c) in sent] for sent in tagged_sents]

def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff
```

We now define a NGramTagChunker class, which will take in tagged sentences as training input, get their WTC triples (word, POS tag, chunk tag), and train a BigramTagger with a UnigramTagger as the backoff tagger. We also define a parse() function to perform shallow parsing on new sentences.

```
from nltk.tag import UnigramTagger, BigramTagger
from nltk.chunk import ChunkParserI

class NGramTagChunker(ChunkParserI):

  def __init__(self, train_sentences,
               tagger_classes=[UnigramTagger, BigramTagger]):
    train_sent_tags = conll_tag_chunks(train_sentences)
    self.chunk_tagger = combined_tagger(train_sent_tags, tagger_classes)

  def parse(self, tagged_sentence):
    if not tagged_sentence:
        return None
    pos_tags = [tag for word, tag in tagged_sentence]
    chunk_pos_tags = self.chunk_tagger.tag(pos_tags)
    chunk_tags = [chunk_tag for (pos_tag, chunk_tag) in chunk_pos_tags]
    wpc_tags = [(word, pos_tag, chunk_tag) for ((word, pos_tag), chunk_tag)
                    in zip(tagged_sentence, chunk_tags)]
    return conlltags2tree(wpc_tags)
```

In this class, the constructor __init__() function is used to train the shallow parser using n-gram tagging based on the WTC triples for each sentence. Internally, it takes a list of training sentences as input, which is annotated with chunked parse tree metadata. It uses the conll_tag_chunks() function, which we defined earlier, to get a list of WTC triples for each chunked parse tree. Finally, it trains a BigramTagger with a Unigram tagger as a backoff tagger using these triples and stores the training model in self.chunk_tagger.

Remember that you can parse other n-gram based taggers for training by using the tagger_classes parameter. Once trained, the parse() function can be used to evaluate the tagger on test data and shallow parse new sentences. Internally, it takes a POS tagged sentence as input, separates the POS tags from the sentence, and uses our trained self.chunk_tagger to get the IOB chunk tags for the sentence. This is then combined with the original sentence tokens and we use the `conlltags2tree()` function to get our final shallow parsed tree. The following snippet shows our parser in action. See Figure 3-11.

```
# train the shallow parser
ntc = NGramTagChunker(train_data)

# test parser performance on test data
print(ntc.evaluate(test_data))

ChunkParse score:
    IOB Accuracy:  97.2%%
    Precision:     91.4%%
    Recall:        94.3%%
    F-Measure:     92.8%%

# parse our sample sentence
sentence_nlp = nlp(sentence)
tagged_sentence = [(word.text, word.tag_) for word in sentence_nlp]
tree = ntc.parse(tagged_sentence)
print(tree)
tree

(S
  (NP US/NNP)
  unveils/VBZ
  (NP world/NN 's/POS most/RBS powerful/JJ supercomputer/NN)
  ,/,
  beats/VBZ
  (NP China/NNP)
  ./.)
```

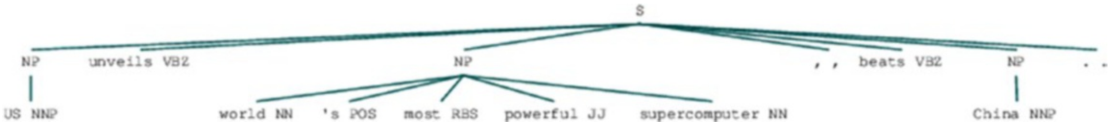


***Figure 3-11.** Shallow parsed news headline using n-gram based chunking on treebank data*

This output depicts our parser performance on the treebank test set data, which has an overall accuracy of 99.6%, which is excellent! Figure 3-11 also shows us how the parse tree looks for our sample news headline.

Let's now train and evaluate our parser on the `conll2000` corpus, which contains excerpts from *The Wall Street Journal* and is a much larger corpus. We will train our parser on the first 10,000 sentences and test its performance on the remaining 940+ sentences. The following snippet depicts this process.

```
from nltk.corpus import conll2000
wsj_data = conll2000.chunked_sents()
train_wsj_data = wsj_data[:10000]
test_wsj_data = wsj_data[10000:]

# look at a sample sentence in the corpus
print(train_wsj_data[10])

(S
  (NP He/PRP)
  (VP reckons/VBZ)
  (NP the/DT current/JJ account/NN deficit/NN)
  (VP will/MD narrow/VB)
  (PP to/TO)
  (NP only/RB #/# 1.8/CD billion/CD)
  (PP in/IN)
  (NP September/NNP)
  ./.)

# train the shallow parser
tc = NGramTagChunker(train_wsj_data)

# test performance on the test data
print(tc.evaluate(test_wsj_data))

ChunkParse score:
    IOB Accuracy:  89.1%%
    Precision:     80.3%%
    Recall:        86.1%%
    F-Measure:     83.1%%
```

This output shows that our parser achieved an overall accuracy of around 89%, which is quite good considering this corpus is much larger compared to the `treebank` corpus. Let's look at how it chunks our sample news headline.

```
# parse our sample sentence
tree = tc.parse(tagged_sentence)
print(tree)
tree
```

```
(S
  (NP US/NNP)
  (VP unveils/VBZ)
  (NP world/NN)
  (NP 's/POS most/RBS powerful/JJ supercomputer/NN)
  ,/,
  (VP beats/VBZ)
  (NP China/NNP)
  ./.)
```

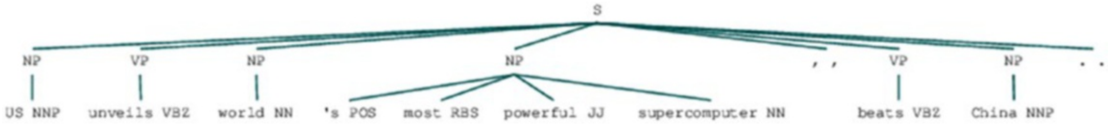


***Figure 3-12.** Shallow parsed news headline using n-gram based chunking on conll2000 data*

Figure 3-12 shows us how the parse tree looks for our sample news headline with more defined verb phrases as compared to previous parse trees. You can also look at implementing shallow parsers using other techniques, like supervised classifiers, by leveraging the `ClassifierBasedTagger` class.

Dependency Parsing

In dependency-based parsing, we try to use dependency-based grammars to analyze and infer both structure and semantic dependencies and relationships between tokens in a sentence. Refer to the "Dependency Grammars" subsection under "Grammar" in the "Language Syntax and Structure" section of Chapter 1 to refresh your memory. Dependency grammars help us annotate sentences with dependency tags, which are one-to-one mappings between tokens signifying dependencies between them. A dependency grammar-based parse tree representation is a labeled and directed tree or graph to be more precise. The nodes are always the lexical tokens and the labeled edges depict dependency relationships between the heads and their dependents. The labels on the edges indicate the grammatical role of the dependent.

The basic principle behind a dependency grammar is that in any sentence in the language, all words except one have some relationship or dependency on other words in the sentence. The word that has no dependency is called the *root* of the sentence. The verb is taken as the root of the sentence in most cases. All the other words are directly or indirectly linked to the root verb using links , which are the dependencies. If we wanted to draw the dependency syntax tree for our sentence, "The brown fox is quick and he is jumping over the lazy dog," we would have the structure depicted in Figure 3-13.

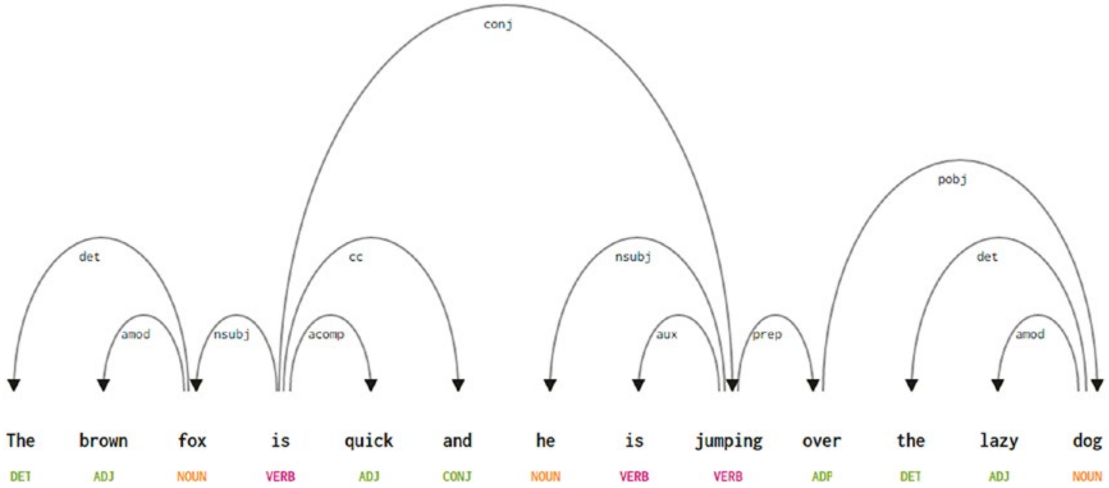


Figure 3-13. *A dependency parse tree for a sample sentence*

These dependency relationships each have their own meanings and are part of a list of universal dependency types. This is discussed in an original paper, entitled "Universal Stanford Dependencies: A Cross-Linguistic Typology," by de Marneffe et al., 2014. You can check out the exhaustive list of dependency types and their meanings at `http://universaldependencies.org/u/dep/index.html`. Just to refresh your memory, if we observe some of these dependencies, it is not too hard to understand them.

- The dependency tag `det` is pretty intuitive—it denotes the determiner relationship between a nominal head and the determiner. Usually, the word with `POS` tag `DET` will also have the `det` dependency tag relation. Examples include fox → the and dog → the.
- The dependency tag `amod` stands for adjectival modifier and stands for any adjective that modifies the meaning of a noun. Examples include fox → brown and dog → lazy.
- The dependency tag `nsubj` stands for an entity that acts as a subject or agent in a clause. Examples include is → fox and jumping → he.
- The dependencies `cc` and `conj` have more to do with linkages related to words connected by coordinating conjunctions. Examples include is → and and is → jumping.
- The dependency tag `aux` indicates the auxiliary or secondary verb in the clause. Example: jumping → is.
- The dependency tag `acomp` stands for adjective complement and acts as the complement or object to a verb in the sentence. Example: is → quick.
- The dependency tag `prep` denotes a prepositional modifier, which usually modifies the meaning of a noun, verb, adjective, or preposition. Usually, this representation is used for prepositions having a noun or noun phrase complement. Example: jumping → over.
- The dependency tag `pobj` is used to denote the object of a preposition. This is usually the head of a noun phrase following a preposition in the sentence. Example: over → dog.

Let's look at some ways in which we can build dependency parsers for parsing unstructured text!

Building Dependency Parsers

We use a couple of state-of-the-art libraries, including NLTK and spaCy, to generate dependency-based parse trees and test them on our sample news headline. *SpaCy* had two types of English dependency parsers based on what language models you use. You can find more details at `https://spacy.io/api/annotation#section-dependency-parsing`.

Based on language models, you can use the Universal Dependencies Scheme or the CLEAR Style Dependency Scheme, also available in NLP4Jnow. We now leverage spaCy and print the dependencies for each token in our news headline.

```
dependency_pattern = '{left}<---{word}[{w_type}]--->{right}\n--------'
for token in sentence_nlp:
    print(dependency_pattern.format(word=token.orth_,
                                  w_type=token.dep_,
                                  left=[t.orth_
                                            for t
                                            in token.lefts],
                                  right=[t.orth_
                                             for t
                                             in token.rights]))
```

```
[]<---US[nsubj]--->[]
--------
['US']<---unveils[ROOT]--->['supercomputer', ',', 'beats', '.']
--------
[]<---world[poss]--->["'s"]
--------
[]<---'s[case]--->[]
--------
[]<---most[advmod]--->[]
--------
['most']<---powerful[amod]--->[]
--------
['world', 'powerful']<---supercomputer[dobj]--->[]
--------
[]<---,[punct]--->[]
--------
[]<---beats[conj]--->['China']
--------
[]<---China[dobj]--->[]
--------
[]<---.[punct]--->[]
--------
```

This output gives us each token and its dependency type. The left arrow points to the dependencies on its left and the right arrow points to the dependencies on its right. It is evident that the verb "beats" is the root since it doesn't have any other dependencies as compared to the other tokens. To learn more about each annotation, you can always refer to the CLEAR dependency scheme at https://emorynlp.github.io/nlp4j/components/dependency-parsing.html. We can also visualize these dependencies in a better way using the following code. See Figure 3-14.

```
from spacy import displacy

displacy.render(sentence_nlp, jupyter=True,
                options={'distance': 110,
                         'arrow_stroke': 2,
                         'arrow_width': 8})
```

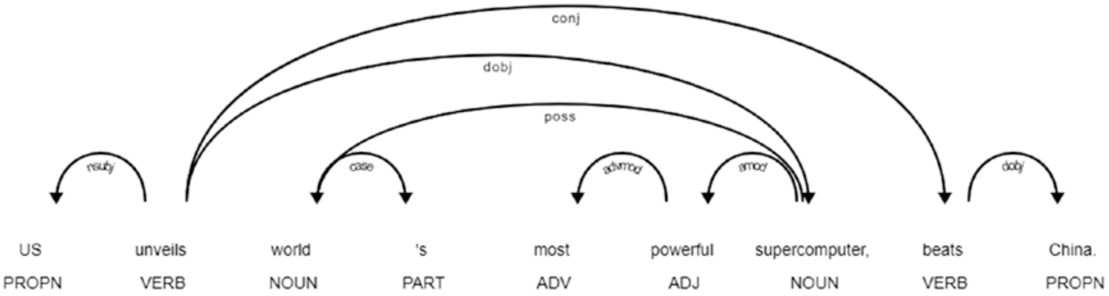


Figure 3-14. *Visualizing our news headline dependency tree using spaCy*

You can also leverage NLTK and the Stanford Dependency Parser to visualize and build the dependency tree. We showcase the dependency tree in its raw and annotated forms. We start by building the annotated dependency tree and showing it using Graphviz. See Figure 3-15.

```
from nltk.parse.stanford import StanfordDependencyParser
sdp = StanfordDependencyParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar', path_to_models_jar='E:/stanford/
stanford-parser-full-2015-04-20/stanford-parser-3.5.2-models.jar')
```

```
# perform dependency parsing
result = list(sdp.raw_parse(sentence))[0]

# generate annotated dependency parse tree
result
```

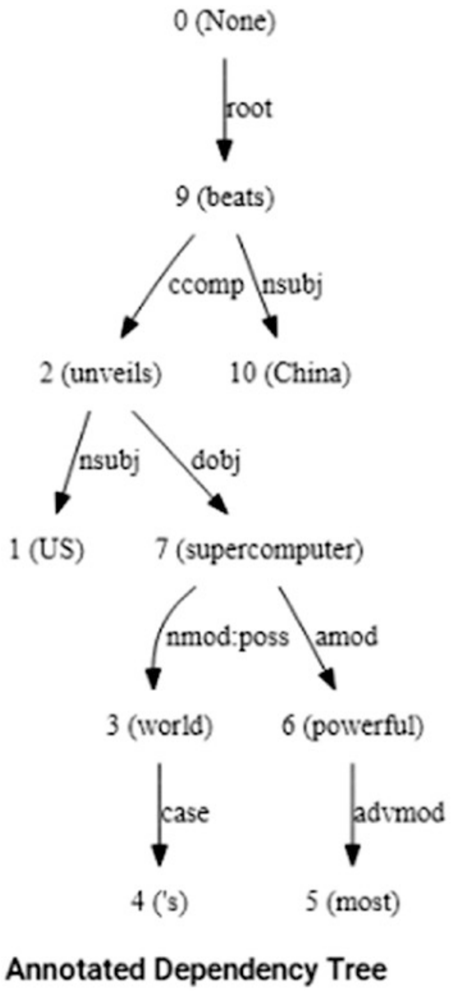


Figure 3-15. *Visualizing our news headline annotated dependency tree using NLTK and the Stanford Dependency Parser*

We can also look at the actual dependency components in the form of triplets using the following code snippet.

```
# generate dependency triples
[item for item in result.triples()]
```

```
[(('beats', 'VBZ'), 'ccomp', ('unveils', 'VBZ')),
 (('unveils', 'VBZ'), 'nsubj', ('US', 'NNP')),
 (('unveils', 'VBZ'), 'dobj', ('supercomputer', 'NN')),
 (('supercomputer', 'NN'), 'nmod:poss', ('world', 'NN')),
 (('world', 'NN'), 'case', ("'s", 'POS')),
 (('supercomputer', 'NN'), 'amod', ('powerful', 'JJ')),
 (('powerful', 'JJ'), 'advmod', ('most', 'RBS')),
 (('beats', 'VBZ'), 'nsubj', ('China', 'NNP'))]
```

This gives us a detailed view into each token and the dependency relationships between tokens. Let's build and visualize the raw dependency tree now. See Figure 3-16.

```
# print simple dependency parse tree
dep_tree = result.tree()
print(dep_tree)
```

```
(beats (unveils US (supercomputer (world 's) (powerful most))) China)
```

```
# visualize simple dependency parse tree
dep_tree
```

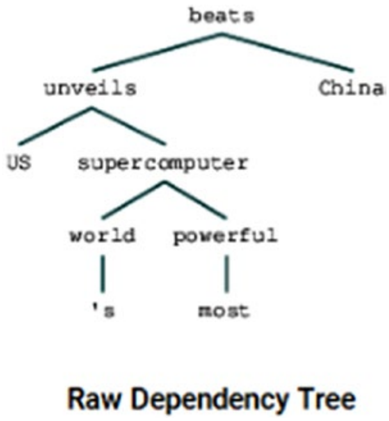


Figure 3-16. *Visualizing our news headline raw dependency tree using NLTK and Stanford Dependency Parser*

Notice the similarities with the tree we obtained earlier in Figure 3-15. The annotations help with understanding the type of dependency among the different tokens. You can also see how easily we can generate dependency parse trees for sentences and analyze and understand relationships and dependencies among the tokens. The Stanford Parser is quite stable and robust. It integrates well with NLTK. We recommend using the NLTK or spaCy parsers, as both of them are quite good.

Constituency Parsing

Constituent based grammars are used to analyze and determine the constituents that a sentence is composed of. Besides determining the constituents, another important objective is to determine the internal structure of these constituents and how they link to each other. There are usually several rules for different types of phrases based on the type of components they can contain and we can use them to build parse trees. Refer to the "Constituency Grammars" subsection under "Grammar" in the "Language Syntax and Structure" section of Chapter 1 to refresh your memory and look at some examples of sample parse trees.

In general, constituency based grammar helps specify how we can break a sentence into various constituents. Once that is done, it helps in breaking down those constituents into further subdivisions; this process repeats until we reach the level of individual tokens or words. Typically, these grammar types can be used to model or represent the internal structure of sentences in terms of a hierarchically ordered structure of their constituents. Each word usually belongs to a specific lexical category in the case and forms the head word of different phrases. These phrases are formed based on rules called phrase structure rules.

Phrase structure rules form the core of constituency grammars, because they talk about syntax and rules that govern the hierarchy and ordering of the various constituents in the sentences. These rules cater to two things primarily:

- They determine what words are used to construct the phrases or constituents.
- They determine how we need to order these constituents.

The generic representation of a phrase structure rule is **S → AB** , which depicts that the structure **S** consists of constituents **A** and **B** , and the ordering is **A** followed by **B** . While there are several rules (refer to Chapter 1 if you want to dive deeper), the most important rule describes how to divide a sentence or a clause. The phrase structure rule denotes a binary division for a sentence or a clause as **S → NP VP**, where **S** is the sentence or clause, and it is divided into the subject, denoted by the noun phrase (**NP**) and the predicate, denoted by the verb phrase (**VP**).

These grammars have various production rules and usually a context free grammar (CFG) or Phrase Structured Grammar is sufficient for this. A constituency parser can be built based on such grammars/rules, which are usually collectively available as context-free grammar (CFG) or phrase-structured grammar. The parser will process input sentences according to these rules and help in building a parse tree. A sample tree is depicted in Figure 3-17.

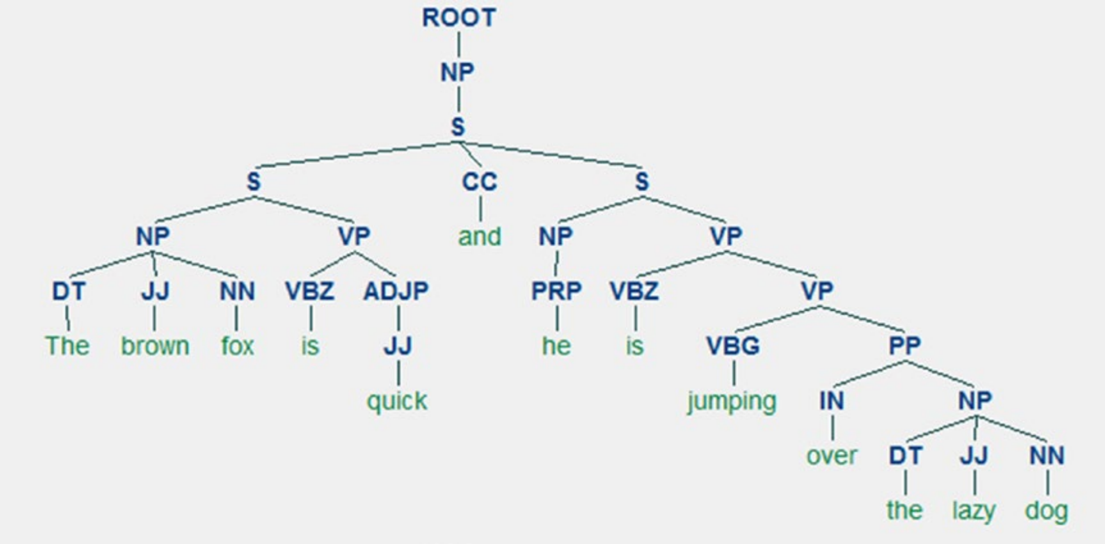


***Figure 3-17.** An example of constituency parsing showing a nested hierarchical structure*

The parser brings the grammar to life and can be said to be a procedural interpretation of the grammar. There are various types of parsing algorithms some of which are mentioned as follows:

- Recursive Descent parsing
- Shift Reduce parsing
- Chart parsing
- Bottom-up parsing

- Top-down parsing
- PCFG parsing

Going through these in detail would be impossible in the current scope. However, NLTK provides some excellent information on them at http://www.nltk.org/book/ch08.html in their official book. We describe some of these parsers briefly and look at PCFG parsing in detail when we implement our own parser later.

- *Recursive Descent parsing* usually follows a top-down parsing approach; it reads in tokens from the input sentence and tries to match them with the terminals from the grammar production rules. It keeps looking ahead by one token and advances the input read pointer each time it gets a match.
- *Shift Reduce parsing* follows a bottom-up parsing approach where it finds sequences of tokens (words/phrases) that correspond to the right side of grammar productions and then replaces it with the left side for that rule. This process continues until the whole sentence is reduced to give us a parse tree.
- *Chart parsing* uses dynamic programming to store intermediate results and reuses them when needed to get significant efficiency gains. In this case, chart parsers store partial solutions and look them up when needed to get to the complete solution.

Building Constituency Parsers

We will be using NLTK and the Stanford Parser to generate parse trees since they are state-of-the-art and work very well.

Prerequisites Download the official Stanford Parser from http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip, which seems to work quite well. You can try a later version by going to http://nlp.stanford.edu/software/lex-parser.shtml#Download and checking the Release History section. After downloading, unzip it to a known location in your filesystem. Once you're done, you are now ready to use the parser from NLTK, which we will be exploring shortly.

The Stanford Parser generally uses a *PCFG (probabilistic context-free grammar) parser*. A PCFG is a context-free grammar that associates a probability with each of its production rules. The probability of a parse tree generated from a PCFG is simply the production of the individual probabilities of the productions used to generate it. Let's put this parser to action now! See Figure 3-18.

```
# set java path
import os
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

# create parser object
from nltk.parse.stanford import StanfordParser
scp = StanfordParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar',
                     path_to_models_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser-3.5.2-models.jar')

# get parse tree
result = list(scp.raw_parse(sentence))[0]
# print the constituency parse tree
print(result)
```

```
(ROOT
  (SINV
    (S
      (NP (NNP US))
      (VP
        (VBZ unveils)
        (NP
          (NP (NN world) (POS 's))
          (ADJP (RBS most) (JJ powerful))
          (NN supercomputer))))
    (, ,)
    (VP (VBZ beats))
    (NP (NNP China))
    (. .)))
```

```
# visualize the parse tree
from IPython.display import display
display(result)
```

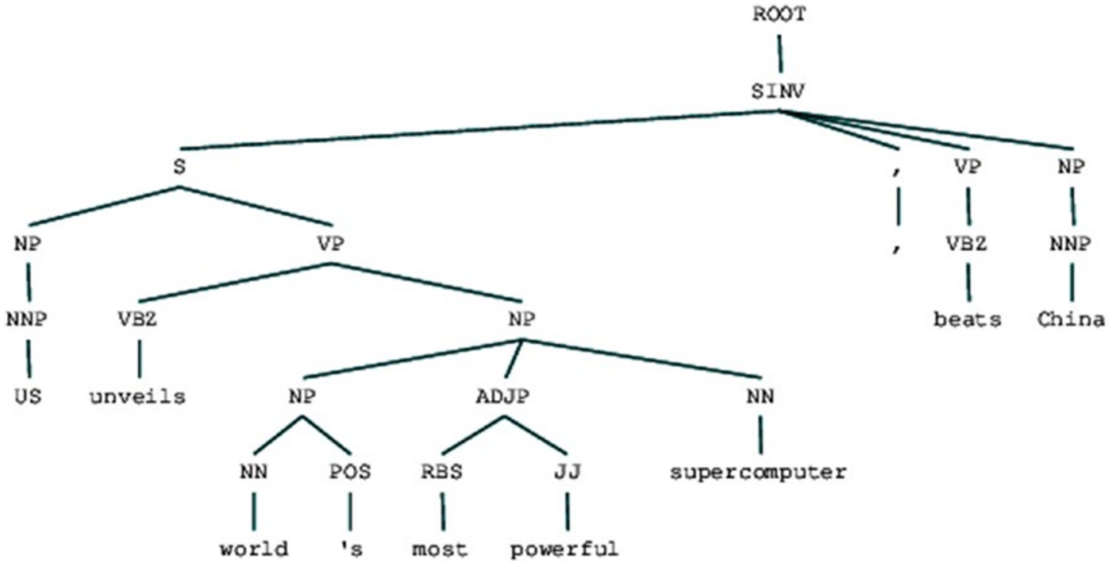


Figure 3-18. *Constituency parsing on our sample news headline*

We can see the nested hierarchical structure of the constituents in the preceding output as compared to the flat structure in shallow parsing. In case you are wondering what SINV means, it represents *an inverted declarative sentence*, i.e. one in which the subject follows the tensed or modal verb. Refer to the "Penn Treebank Reference" at `https://web.archive.org/web/20130517134339/http://bulba.sdsu.edu/jeanette/thesis/PennTags.html` as needed to look up other tags.

There are various ways that you can build your own constituency parsers, including creating your own CFG production rules and then using a parser to use that grammar. To build your own CFG, you can use the `nltk.CFG.fromstring` function to feed in your own production rules and then use parsers like `ChartParser` or `RecursiveDescentParser`, which both belong to the NLTK package. Feel free to build some toy grammars and play around with these parsers.

We now look at a way to build a constituency parser that scales well and is efficient. The problem with regular CFG parsers like chart and recursive descent parsers is that they can get easily overwhelmed by the sheer number of total possible parses and can become extremely slow. This is where weighted grammars like PCFG (Probabilistic Context Free Grammar) and probabilistic parsers like the Viterbi parser prove to be more effective.

A PCFG is a context free grammar that associates a probability with each of its production rules. The probability of a parse tree generated from a PCFG is simply the production of the individual probabilities of the productions that were used to generate it. We will use NLTK's `ViterbiParser` to train a parser on the `treebank` corpus, which provides annotated parse trees for each sentence in the corpus. This parser is a bottom-up PCFG parser that uses dynamic programming to find the most likely parse at each step. We start our process of building our own parser by loading the necessary training data and dependencies.

```
import nltk
from nltk.grammar import Nonterminal
from nltk.corpus import treebank
# load and view training data
training_set = treebank.parsed_sents()
print(training_set[1])
```

```
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
      (NP (NN chairman))
      (PP
        (IN of)
        (NP
          (NP (NNP Elsevier) (NNP N.V.))
          (, ,)
          (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
  (. .))
```

Now we build the production rules for our grammar by extracting the productions from the tagged and annotated training sentences and adding them.

```
# extract the productions for all annotated training sentences
treebank_productions = list(
                        set(production
                            for sent in training_set
                            for production in sent.productions()
                        )
                    )
# view some production rules
treebank_productions[0:10]
```

```
[VP -> VB NP-2 PP-CLR ADVP-MNR,
 NNS -> 'foods',
 NNP -> 'Joanne',
 JJ -> 'state-owned',
 VP -> VBN PP-LOC,
 NN -> 'turmoil',
 SBAR -> WHNP-240 S,
 QP -> DT VBN CD TO CD,
 NN -> 'cultivation',
 NNP -> 'Graham']
```

```
# add productions for each word, POS tag
for word, tag in treebank.tagged_words():
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
        treebank_productions.append(production)
```

```
# build the PCFG based grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'), treebank_
productions)
```

Now that we have our necessary grammar with production rules, we create our parser using the following snippet by training it on the grammar and then try to evaluate it on our sample news headline.

```
# build the parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)

# get sample sentence tokens
tokens = nltk.word_tokenize(sentence)

# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-87-2b0fd95b2fbd> in <module>()
     16
     17 # get parse tree for sample sentence
---> 18 result = list(viterbi_parser.parse(tokens))

ValueError: Grammar does not cover some of the input words: "'unveils',
'beats'".
```

Unfortunately, we get an error when we try to parse our sample sentence tokens with our newly built parser. The reason is quite clear from the error that some of the words in our sample sentence are not covered by the Treebank-based grammar because they are not present in our treebank corpus. Since this constituency based grammar uses POS tags and phrase tags to build the tree based on the training data, we will add the token and POS tags for our sample sentence in our grammar and rebuild the parser.

```
# get tokens and their POS tags and check it
tagged_sent = nltk.pos_tag(nltk.word_tokenize(sentence))
print(tagged_sent)
```

```
[('US', 'NNP'), ('unveils', 'JJ'), ('world', 'NN'), ("'s", 'POS'), ('most',
'RBS'), ('powerful', 'JJ'), ('supercomputer', 'NN'), (',', ','), ('beats',
'VBZ'), ('China', 'NNP'), ('.', '.')]
```

```
# extend productions for sample sentence tokens
for word, tag in tagged_sent:
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
        treebank_productions.append(production)
```

```
# rebuild grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)
# rebuild parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)
# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))[0]

# print parse tree
print(result)

(S
  (NP-SBJ-2
    (NP (NNP US))
    (NP
      (NP (JJ unveils) (NN world) (POS 's))
      (JJS most)
      (JJ powerful)
      (NN supercomputer)))
  (, ,)
  (VP (VBZ beats) (NP-TTL (NNP China)))
  (. .)) (p=5.08954e-43)

# visualize parse tree
result
```

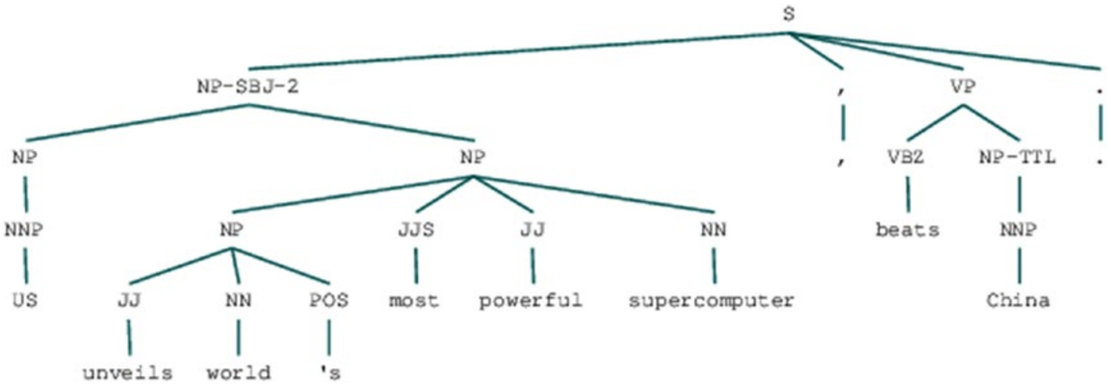


***Figure 3-19.** Constituency parse tree for our sample news headline based on Treebank annotations*

We are now able to successfully generate the parse tree for our sample news headline. You can see the visual representation of the tree in Figure 3-19. Remember that this is a probabilistic PCFG parser and you can see the overall probability of this tree mentioned in the output when we printed our parse tree. The notations of the tags followed here are all based on the Treebank annotations discussed earlier. Thus, this shows us how to build our own constituency-based parser.

Summary

We have covered a lot concepts, techniques, and implementations with regard to text processing and wrangling, syntactic analysis, and understanding text data. A lot of the concepts from Chapter 1 should seem more relevant and clear now that we have implemented them on real examples. The content covered in this chapter is two-fold.

We looked at concepts related to text processing and wrangling. You now know the importance of processing and normalizing text and, as we move on to future chapters, you will see why it becomes more and more important to have well processed and standardized textual data. We have covered a wide variety of techniques for wrangling including text cleaning and tokenization, removing special characters, case conversions, and expanding contractions. We also looked at techniques for correcting text, like spelling corrections. We also built our own spelling corrector and contraction expander in the same context. We found a way to leverage WordNet and correct words with repeated characters. Finally, we looked at various stemming and lemmatization techniques and learned about ways to remove irrelevant words, known as *stopwords*.

The next part of our chapter was dedicated to analyzing and understanding text syntax and structure. We revisited concepts from Chapter 1, including POS tagging, shallow parsing, dependency parsing, and constituency parsing. You now know how to use taggers and parsers on real-world textual data and how to implement your own taggers and parsers. We dive more into analyzing and deriving insights from text in future chapters using various machine learning techniques including classification, clustering, and summarization. Stay tuned!

CHAPTER 4

Feature Engineering for Text Representation

In the previous chapters, we saw how to understand, process, and wrangle text data. However, all machine learning or deep learning models are limited because they cannot understand text data directly and they only understand numeric representations of features as inputs. In this chapter, we look at how to work with text data, which is definitely one of the most abundant sources of unstructured data. Text data usually consists of documents that can represent words, sentences, or even paragraphs of free-flowing text. The inherent lack of structure (no neatly formatted data columns!) and noisy nature of textual data makes it harder for machine learning methods to directly work on raw text data. Hence, in this chapter, we follow a hands-on approach to exploring some of the most popular and effective strategies for extracting meaningful features from text data. These features can then be used to represent text efficiently, which can be further leveraged in building machine learning or deep learning models easily to solve complex tasks.

Feature engineering is very important and is often known as the secret sauce to creating superior and better performing machine learning models. Just one excellent feature could be your ticket to winning a Kaggle challenge or getting more returns based on your forecast! Feature engineering is even more important for unstructured, textual data because we need to convert free-flowing text into some numeric representations, which can then be understood by machine learning algorithms. Even with the advent of automated feature engineering capabilities, you still need to understand the core concepts behind different feature engineering strategies before applying them as black box models. Always remember, “If you are given a box of tools to repair a house, you should know when to use a power drill and when to use a hammer!”.

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_4

In this chapter, we cover a wide variety of techniques for feature engineering to represent text data. We look at traditional models as well as newer models based on deep learning. We cover the following techniques in this chapter:

- Bag of Words model
- Bag of N-Grams model
- TF-IDF model
- Similarity features
- Topic models
- Word2Vec
- GloVe
- FastText

We look at important concepts pertaining to each feature engineering technique and learn how the model is used to represent text data. We also showcase full-fledged hands-on examples, because learning by doing works best! All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Understanding Text Data

I'm sure must have a fair idea of what textual data is, considering we covered three chapters on it! Do remember you can always have text data in the form of structured data attributes, but usually those fall under the umbrella of structured, categorical data. In this scenario, we are talking about free-flowing text in the form of words, phrases, sentences, and entire documents. Essentially, we do have some syntactic structure. Words make phrases, which in turn make sentences, which in turn make paragraphs. However, there is no inherent structure to text documents because you can have a wide variety of words that can vary across documents and each sentence will also be of variable length as compared to a fixed number of data dimensions in structured datasets. This chapter is a perfect example of text data! An important question might be how can we represent text data so it's easy for machines to comprehend and understand?

A *vector space model* is a useful concept when dealing with textual data and is very popular in information retrieval and document ranking. The vector space model is also called the *term vector model* and is defined as a mathematical and algebraic model for transforming and representing text documents as numeric vectors of specific terms, which form the vector dimensions. Mathematically, this can be defined as follows. Consider we have a document **D** in a document vector space **VS**. The number of dimensions or columns for each document will be the total number of distinct terms or words for all documents in the vector space. Hence the vector space can be denoted as follows:

$$VS = \{W_1, W_2, \ \dots , W_n\}$$

where there are **n** distinct words across all documents. Now we can represent document **D** in this vector space as follows:

$$D = \{w_{D1}, w_{D2}, \ \dots , w_{Dn}\}$$

where w_{Dn} denotes the weight for word **n** in document **D**. This weight is a numeric value and can be anything ranging from the frequency of that word in the document, the average frequency of occurrence, embedding weights, or even the *TF-IDF weight*, which we discuss shortly.

An important point to remember about feature extraction and engineering is that once we build a feature engineering model using transformations and mathematical operations, we need to make sure we use the same process when extracting features from new documents to be predicted and not rebuild the whole algorithm again based on the new documents.

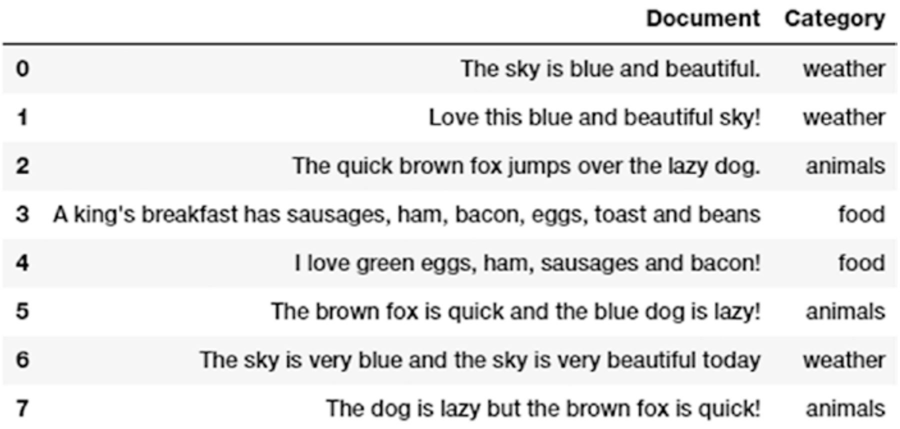
Building a Text Corpus

We need a text corpus to work on and demonstrate different feature engineering and representation methodologies. To keep things simple and easy to understand, we build a simple text corpus in this section. To get started, load the following dependencies in your Jupyter notebook.

```
import pandas as pd
import numpy as np
import re
import nltk
import matplotlib.pyplot as plt

pd.options.display.max_colwidth = 200
%matplotlib inline
```

Let's now build a sample corpus of documents on which we will run most of our analyses in this chapter. A *corpus* is typically a collection of text documents usually belonging to one or more subjects or topics. The following code helps us create this corpus. You can see this sample text corpus in the output in Figure 4-1.

```
# building a corpus of documents
corpus = ['The sky is blue and beautiful.',
          'Love this blue and beautiful sky!',
          'The quick brown fox jumps over the lazy dog.',
          "A king's breakfast has sausages, ham, bacon, eggs, toast and beans",
          'I love green eggs, ham, sausages and bacon!',
          'The brown fox is quick and the blue dog is lazy!',
          'The sky is very blue and the sky is very beautiful today',
          'The dog is lazy but the brown fox is quick!'
]
labels = ['weather', 'weather', 'animals', 'food', 'food', 'animals',
'weather', 'animals']

corpus = np.array(corpus)
corpus_df = pd.DataFrame({'Document': corpus, 'Category': labels})
corpus_df = corpus_df[['Document', 'Category']]
corpus_df
```



	Document	Category
0	The sky is blue and beautiful.	weather
1	Love this blue and beautiful sky!	weather
2	The quick brown fox jumps over the lazy dog.	animals
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food
4	I love green eggs, ham, sausages and bacon!	food
5	The brown fox is quick and the blue dog is lazy!	animals
6	The sky is very blue and the sky is very beautiful today	weather
7	The dog is lazy but the brown fox is quick!	animals

Figure 4-1. *Our sample text corpus*

Figure 4-1 shows that we have taken a few sample text documents belonging to different categories for our toy corpus. Before we talk about feature engineering, we need to do some data preprocessing and wrangling to remove unnecessary characters, symbols, and tokens.

Preprocessing Our Text Corpus

There can be multiple ways of cleaning and preprocessing textual data. In the following points, we highlight some of the most important ones that are used heavily in Natural Language Processing (NLP) pipelines. A lot of this will be a refresher if you have read Chapter 3.

- **Removing tags:** Our text often contains unnecessary content like HTML tags, which do not add much value when analyzing text. The BeautifulSoup library does an excellent job in providing necessary functions for this.

- **Removing accented characters:** In any text corpus, especially if you are dealing with the English language, you might be dealing with accented characters/letters. Hence, you need to make sure that these characters are converted and standardized into ASCII characters. A simple example is converting é to e.

- **Expanding contractions:** In the English language, contractions are basically shortened versions of words or syllables, created by removing specific letters and sounds. Examples include *do not* to *don't* and *I would* to *I'd*. Converting each contraction to its expanded, original form often helps with text standardization.
- **Removing special characters:** Special characters and symbols that are usually non alphanumeric characters often add to the extra noise in unstructured text. More often than not, simple regular expressions (regexes) can be used to achieve this.
- **Stemming and lemmatization:** Word stems are the base form of possible words that can be created by attaching affixes like prefixes and suffixes to the stem to create new words. This is known as inflection. The reverse process of obtaining the base form of a word is known as *stemming*. A simple example are the words *watches*, *watching*, and *watched*. They have the word root stem *watch* as the base form. *Lemmatization* is very similar to stemming, where we remove word affixes to get to the base form of a word. However, the base form in this case is known as the root word but not the root stem. The difference being that the root word is always a lexicographically correct word (present in the dictionary) but the root stem may not always be correct.
- **Removing stopwords:** Words that have little or no significance, especially when constructing meaningful features from text, are known as stopwords. These are usually words that end up having the maximum frequency if you do a simple term or word frequency in a corpus. Words like "a," "an," "the," and so on are considered to be stopwords. There is no universal stopword list, but we use a standard English language stopwords list from NLTK. You can also add your own domain specific stopwords as needed.

You can also do other standard operations like tokenization, removing extra whitespace, text lowercasing and more advanced operations like spelling corrections, grammatical error corrections, removing repeated characters, and so on. If you are interested, check out the detailed section on text preprocessing and wrangling in Chapter 3.

Since the focus of this article is on feature engineering, we build a simple text preprocessor that focuses on removing special characters, extra whitespace, digits, stopwords, and then lowercasing the text corpus.

```
wpt = nltk.WordPunctTokenizer()
stop_words = nltk.corpus.stopwords.words('english')

def normalize_document(doc):
    # lowercase and remove special characters\whitespace
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = wpt.tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)
```

Once we have our basic preprocessing pipeline ready, let's apply it to our sample corpus so we can use it for feature engineering.

```
norm_corpus = normalize_corpus(corpus)
norm_corpus

array(['sky blue beautiful', 'love blue beautiful sky',
       'quick brown fox jumps lazy dog',
       'kings breakfast sausages ham bacon eggs toast beans',
       'love green eggs ham sausages bacon',
       'brown fox quick blue dog lazy', 'sky blue sky beautiful today',
       'dog lazy brown fox quick'], dtype='<U51')
```

This output should give you a clear view of how each of our sample documents look after preprocessing. Let's explore various feature engineering techniques now!

Traditional Feature Engineering Models

Traditional (count-based) feature engineering strategies for textual data belong to a family of models popularly known as the Bag of Words model. This includes term frequencies, TF-IDF (term frequency-inverse document frequency), N-grams, topic models, and so on. While they are effective methods for extracting features from text, due to the inherent nature of the model being just a bag of unstructured words, we lose additional information like the semantics, structure, sequence, and context around nearby words in each text document. There are more advanced models that take care of these aspects and we cover them in a subsequent section in this chapter. The traditional feature engineering models are built using mathematical and statistical methodologies. We look at some of these models and apply them to our sample corpus.

Bag of Words Model

This is perhaps the most simple vector space representational model for unstructured text. A vector space model is simply a mathematical model to represent unstructured text (or any other data) as numeric vectors, such that each dimension of the vector is a specific feature/attribute. The Bag of Words model represents each text document as a numeric vector where each dimension is a specific word from the corpus and the value could be its frequency in the document, occurrence (denoted by 1 or 0), or even weighted values. The model's name is such because each document is represented literally as a bag of its own words, disregarding word order, sequences, and grammar.

```
from sklearn.feature_extraction.text import CountVectorizer
# get bag of words features in sparse format
cv = CountVectorizer(min_df=0., max_df=1.)
cv_matrix = cv.fit_transform(norm_corpus)
cv_matrix
```

```
<8x20 sparse matrix of type '<class 'numpy.int64'>'
      with 42 stored elements in Compressed Sparse Row format>
```

```
# view non-zero feature positions in the sparse matrix
print(cv_matrix)
```

```
  (0, 2)     1
  (0, 3)     1
  (0, 17)    1
  (1, 14)    1
  ...
  ...
  (6, 17)    2
  (7, 6)     1
  (7, 13)    1
  (7, 8)     1
  (7, 5)     1
  (7, 15)    1
```

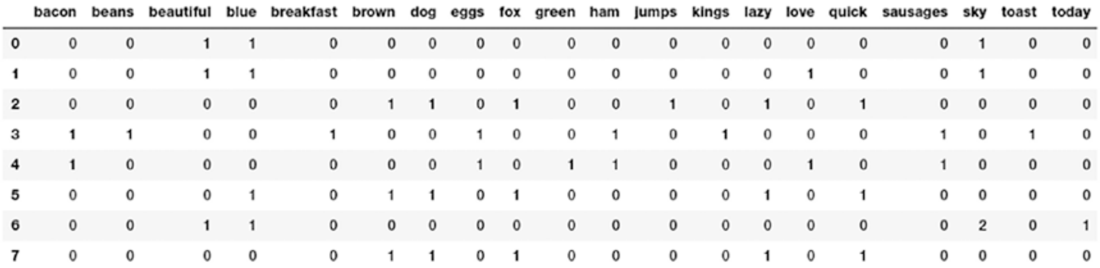
The feature matrix is traditionally represented as a sparse matrix since the number of features increases phenomenally with each document considering each distinct word becomes a feature. The preceding output tells us the total count for each (**x, y**) pair. Here, **x** represents a document and **y** represents a specific word/feature and the value is the number of times **y** occurs in **x**. We can leverage the following code to view the output in a dense matrix representation.

```
# view dense representation
# warning might give a memory error if data is too big
cv_matrix = cv_matrix.toarray()
cv_matrix
```

```
array([[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0],
       [1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]],
       dtype=int64)
```

Thus, you can see that these documents have been converted into numeric vectors so that each document is represented by one vector (row) in the feature matrix and each column represents a unique word as a feature. The following code represents this in a more easy to understand format. See Figure 4-2.

```
# get all unique words in the corpus
vocab = cv.get_feature_names()
# show document feature vectors
pd.DataFrame(cv_matrix, columns=vocab)
```



	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
2	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0
3	1	1	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0
4	1	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1
7	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0

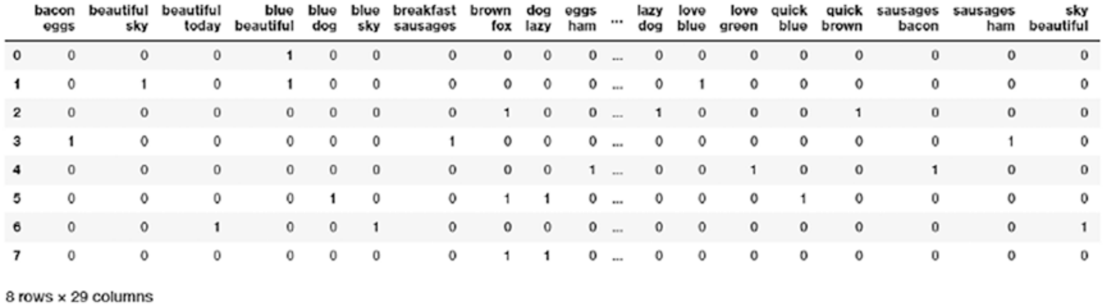
Figure 4-2. *Our Bag of Words model based document feature vectors*

Figure 4-2 should make things more clear! You can clearly see that each column or dimension in the feature vectors represents a word from the corpus and each row represents one of our documents. The value in any cell represents the number of times that word (represented by column) occurs in the specific document (represented by row). A simple example would be the first document has the words blue, beautiful, and sky occurring once each and hence the corresponding features have a value of 1 for the first row in the preceding output. Hence, if a corpus of documents consists of N unique words across all the documents, we would have an N-dimensional vector for each of the documents.

Bag of N-Grams Model

A word is just a single token, often known as a *unigram* or *1-gram*. We already know that the Bag of Words model doesn't consider the order of words. But what if we also wanted to take into account phrases or collection of words that occur in a sequence? N-grams help us do that. An N-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence. Bi-grams indicate n-grams of order 2 (two words), tri-grams indicate n-grams of order 3 (three words), and so on. The Bag of N-Grams model is just an extension of the Bag of Words model that leverages N-gram based features. The following example depicts bi-gram based features in each document feature vector. See Figure 4-3.

```
# you can set the n-gram range to 1,2 to get unigrams as well as bigrams
bv = CountVectorizer(ngram_range=(2,2))
bv_matrix = bv.fit_transform(norm_corpus)

bv_matrix = bv_matrix.toarray()
vocab = bv.get_feature_names()
pd.DataFrame(bv_matrix, columns=vocab)
```



	bacon eggs	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	breakfast sausages	brown fox	dog lazy	eggs ham	...	lazy dog	love blue	love green	quick blue	quick brown	sausages bacon	sausages ham	sky beautiful
0	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	...	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	...	1	0	0	0	1	0	0	0
3	1	0	0	0	0	0	1	0	0	0	...	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	1	...	0	0	1	0	0	1	0	0
5	0	0	0	0	1	0	0	1	1	0	...	0	0	0	1	0	0	0	0
6	0	0	1	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	1	1	0	...	0	0	0	0	0	0	0	0

8 rows × 29 columns

Figure 4-3. *Bi-gram based feature vectors using the Bag of N-Grams model*

This gives us feature vectors for our documents, where each feature consists of a bi-gram representing a sequence of two words and values represent how many times the bi-gram was present for our documents. We encourage you to play around with the `ngram_range` argument. Try out these functions by setting `ngram_range` to `(1, 3)` and see the outputs!

TF-IDF Model

There are some potential problems that might arise with the Bag of Words model when it is used on large corpora. Since the feature vectors are based on absolute term frequencies, there might be some terms that occur frequently across all documents and these may tend to overshadow other terms in the feature set. Especially words that don't occur as frequently, but might be more interesting and effective as features to identify specific categories. This is where TF-IDF comes into the picture. TF-IDF stands for *term frequency-inverse document frequency.* It's a combination of two metrics, term frequency (tf) and inverse document frequency (idf). This technique was originally developed as a metric for ranking search engine results based on user queries and has come to be a part of information retrieval and text feature extraction.

Let's formally define TF-IDF now and look at the mathematical representations before diving into its implementation. Mathematically, TD-IDF is the product of two metrics and can be represented as follows:

$$tfidf = tf \times idf$$

where term frequency (*tf*) and inverse-document frequency (`idf`) represent the two metrics we just talked about. *Term frequency,* denoted by tf, is what we computed in the Bag of Words model in the previous section. Term frequency in any document vector is denoted by the raw frequency value of that term in a particular document. Mathematically it can be represented as follows:

$$tf\left(w,D\right) = f_{w_D}$$

where f_{w_D} denoted frequency for word **w** in document **D**, which becomes the term frequency (tf). Sometimes you can also normalize the absolute raw frequency using logarithms or averaging the frequency. We use the raw frequency in our computations.

Inverse document frequency denoted by `idf` is the inverse of the document frequency for each term and is computed by dividing the total number of documents in our corpus by the document frequency for each term and then applying logarithmic scaling to the result. In our implementation, we will be adding 1 to the document frequency for each term to indicate that we also have one more document in our corpus, which essentially has every term in the vocabulary. This is to prevent potential division by zero errors and smoothen the inverse document frequencies. We also add 1 to the result of our `idf` computation to avoid ignoring terms that might have zero `idf`. Mathematically, our implementation for `idf` can be represented as follows:

$$idf\left(w,D\right) = 1 + \log\frac{N}{1 + df\left(w\right)}$$

where $idf(w, D)$ represents the `idf` for the term/word ***w in*** document **D**, **N** represents the total number of documents in our corpus, and $df(t)$ represents the number of documents in which the term ***w*** is present.

Thus, the term frequency-inverse document frequency can be computed by multiplying these two measures. The final TF-IDF metric that we will be using is a normalized version of the `tfidf` matrix that we get from the product of `tf` and `idf`. We will normalize the `tfidf` matrix by dividing it by the L2 norm of the matrix, also known as the Euclidean norm, which is the square root of the sum of the square of each term's `tfidf` weight. Mathematically we can represent the final `tfidf` feature vector as follows:

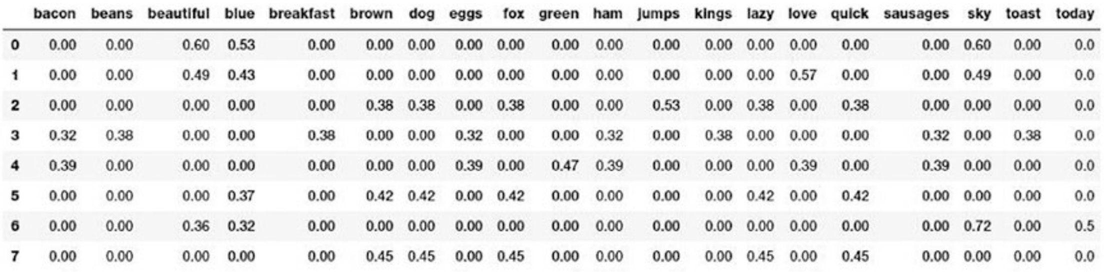
$$tfidf = \frac{tfidf}{\| tfidf \|}$$

where $\|tfidf\|$ represents the Euclidean L2 norm for the `tfidf` matrix. There are multiple variants of this model but they all end up with similar results. Let's apply this on our corpus now!

Using TfidfTransformer

The following code shows an implementation of getting the **tfidf**-based feature vectors considering we already have our Bag of Words feature vectors from a previous section. See Figure 4-4.

```
from sklearn.feature_extraction.text import TfidfTransformer

tt = TfidfTransformer(norm='l2', use_idf=True)
tt_matrix = tt.fit_transform(cv_matrix)

tt_matrix = tt_matrix.toarray()
vocab = cv.get_feature_names()
pd.DataFrame(np.round(tt_matrix, 2), columns=vocab)
```



	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

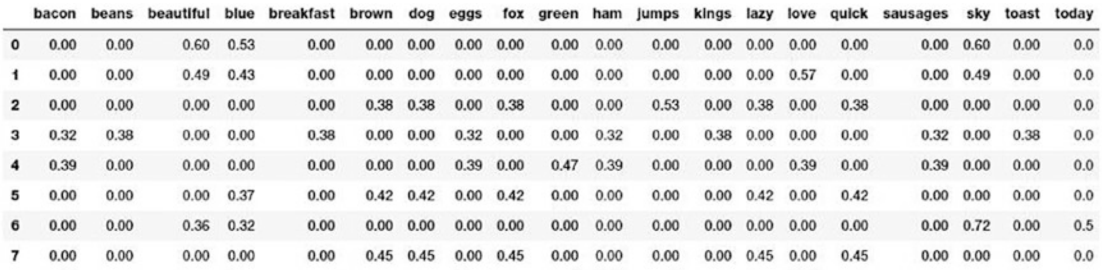
Figure 4-4. *Our TF-IDF model based document feature vectors using TfidfTransformer*

You can see that we used the L2 norm option in the parameters and made sure we smoothen the IDFs to give weight to terms that may have zero IDF so that we do not ignore them.

Using TfidfVectorizer

You don't always need to generate features beforehand using a Bag of Words or count based model before engineering TF-IDF features. The TfidfVectorizer by Scikit-Learn enables us to directly compute the `tfidf` vectors by taking the raw documents as input and internally computing the term frequencies as well as the inverse document frequencies. This eliminates the need to use `CountVectorizer` to compute the term frequencies based on the Bag of Words model. Support is also present for adding n-grams to the feature vectors. We can see the function in action in the following snippet. See Figure 4-5.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tv = TfidfVectorizer(min_df=0., max_df=1., norm='l2',
                     use_idf=True, smooth_idf=True)
tv_matrix = tv.fit_transform(norm_corpus)
tv_matrix = tv_matrix.toarray()

vocab = tv.get_feature_names()
pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```



	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

Figure 4-5. *Our TF-IDF model based document feature vectors using TfidfVectorizer*

You can see that, just like before, we used the L2 norm option in the parameters and made sure we smoothened the `idfs`. You can see from the output that the `tfidf` feature vectors match to the ones we obtained previously.

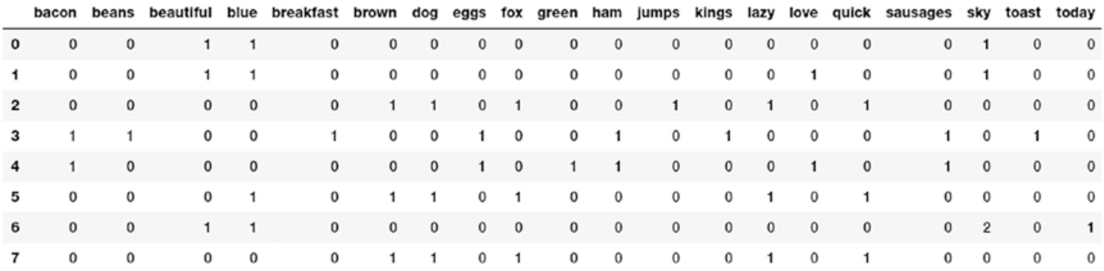
Understanding the TF-IDF Model

This section is dedicated to machine learning experts and our curious readers who are often interested in how things work behind the scenes! We start by loading the necessary dependencies and computing the term frequencies (*TF*) for our sample corpus. See Figure 4-6.

```
# get unique words as feature names
unique_words = list(set([word for doc in [doc.split() for doc in norm_corpus]
                         for word in doc]))
def_feature_dict = {w: 0 for w in unique_words}
print('Feature Names:', unique_words)
print('Default Feature Dict:', def_feature_dict)
```

```
Feature Names: ['lazy', 'fox', 'love', 'jumps', 'sausages', 'blue', 'ham',
'beautiful', 'brown', 'kings', 'eggs', 'quick', 'bacon', 'breakfast',
'toast', 'beans', 'green', 'today', 'dog', 'sky']
Default Feature Dict: {'lazy': 0, 'fox': 0, 'kings': 0, 'love': 0, 'jumps':
0, 'sausages': 0, 'breakfast': 0, 'today': 0, 'brown': 0, 'ham': 0,
'beautiful': 0, 'green': 0, 'eggs': 0, 'blue': 0, 'bacon': 0, 'toast': 0,
'beans': 0, 'dog': 0, 'sky': 0, 'quick': 0}
```

```
from collections import Counter
# build bag of words features for each document - term frequencies
bow_features = []
for doc in norm_corpus:
    bow_feature_doc = Counter(doc.split())
    all_features = Counter(def_feature_dict)
    bow_feature_doc.update(all_features)
    bow_features.append(bow_feature_doc)

bow_features = pd.DataFrame(bow_features)
bow_features
```

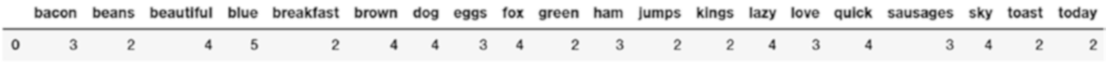


	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
2	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0
3	1	1	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	1	0
4	1	0	0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	1
7	0	0	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0

Figure 4-6. *Constructing count-based Bag of Words features from scratch for our corpus*

We now compute our document frequencies (*DF*) for each term based on the number of documents in which the term occurs. The following snippet shows how to obtain it from our Bag of Words features. See Figure 4-7.

```
import scipy.sparse as sp
feature_names = list(bow_features.columns)

# build the document frequency matrix
df = np.diff(sp.csc_matrix(bow_features, copy=True).indptr)
df = 1 + df # adding 1 to smoothen idf later

# show smoothened document frequencies
pd.DataFrame([df], columns=feature_names)
```



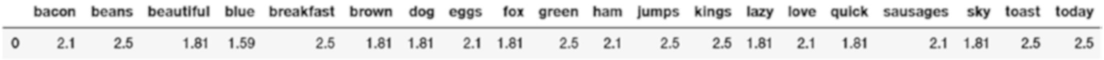
	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	3	2	4	5	2	4	4	3	4	2	3	2	2	4	3	4	3	4	2	2

Figure 4-7. *Document frequencies for each feature in our corpus*

This tells us the document frequency (*DF*) for each term and you can verify it with the documents in our sample corpus. Remember that we added 1 to each frequency value to smoothen the IDF values later and prevent division by zero errors by assuming we have a document (imaginary) that has all the terms once. Thus, if you check in the corpus, you will see that "bacon" occurs 2(+1) times, "sky" occurs 3(+1) times, and so on considering (+1) for our smoothening.

Now that we have the document frequencies, we compute the inverse document frequency (IDF) by using our formula, which we defined earlier. Remember to add 1 to the total count of documents in the corpus to add the document, which we had assumed earlier to contain all the terms at least once for smoothening the idfs. See Figure 4-8.

```
# compute inverse document frequencies
total_docs = 1 + len(norm_corpus)
idf = 1.0 + np.log(float(total_docs) / df)

# show smoothened idfs
pd.DataFrame([np.round(idf, 2)], columns=feature_names)
```

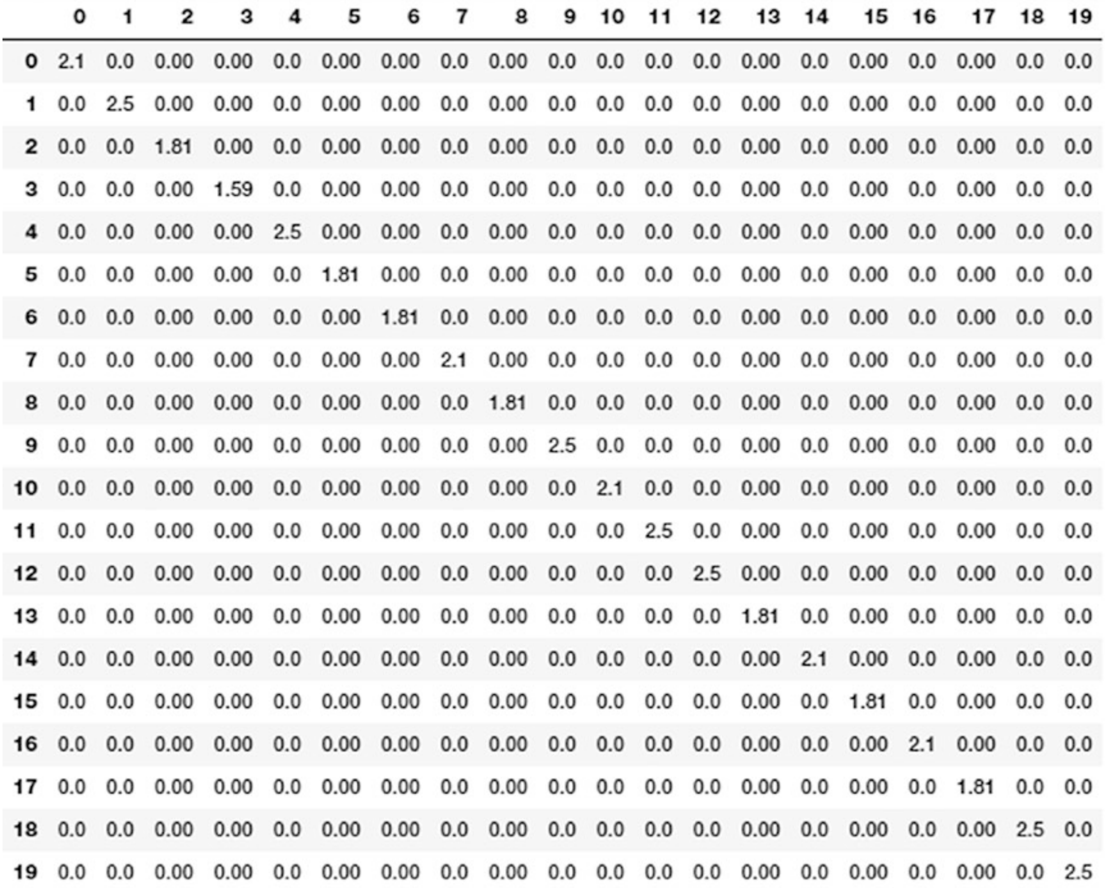


	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	2.1	2.5	1.81	1.59	2.5	1.81	1.81	2.1	1.81	2.5	2.1	2.5	2.5	1.81	2.1	1.81	2.1	1.81	2.5	2.5

Figure 4-8. *Inverse document frequencies for each feature in our corpus*

Thus, we can see that Figure 4-8 depicts the inverse document frequencies (smoothed) for each feature in our corpus. We now convert this into a matrix for easier operations when we compute the overall TF-IDF score later. See Figure 4-9.

```
# compute idf diagonal matrix
total_features = bow_features.shape[1]
idf_diag = sp.spdiags(idf, diags=0, m=total_features, n=total_features)
idf_dense = idf_diag.todense()

# print the idf diagonal matrix
pd.DataFrame(np.round(idf_dense, 2))
```

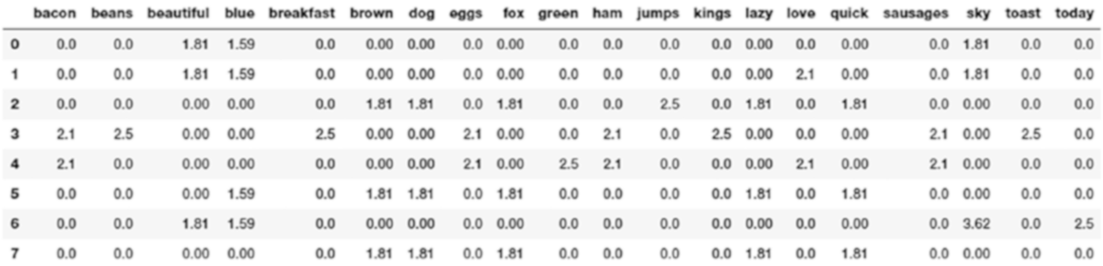


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2.1	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
1	0.0	2.5	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
2	0.0	0.0	1.81	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
3	0.0	0.0	0.00	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
4	0.0	0.0	0.00	0.00	2.5	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
5	0.0	0.0	0.00	0.00	0.0	1.81	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
6	0.0	0.0	0.00	0.00	0.0	0.00	1.81	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
7	0.0	0.0	0.00	0.00	0.0	0.00	0.00	2.1	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
8	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	1.81	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
9	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	2.5	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
10	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	2.1	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
11	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	2.5	0.0	0.00	0.0	0.00	0.0	0.00	0.0	0.0
12	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	2.5	0.00	0.0	0.00	0.0	0.00	0.0	0.0
13	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	1.81	0.0	0.00	0.0	0.00	0.0	0.0
14	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	2.1	0.00	0.0	0.00	0.0	0.0
15	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	1.81	0.0	0.00	0.0	0.0
16	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	2.1	0.00	0.0	0.0
17	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	1.81	0.0	0.0
18	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	2.5	0.0
19	0.0	0.0	0.00	0.00	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.00	0.0	2.5

Figure 4-9. *Constructing an n-verse document frequency diagonal matrix for each feature in our corpus*

You can now see the idf matrix that we created based on our mathematical equation. We also convert it to a diagonal matrix, which will be helpful later when we want to compute the product with term frequency. Now that we have our TFs and IDFs, we can compute the raw TF-IDF feature matrix using matrix multiplication, as depicted in the following snippet. See Figure 4-10.

```
# compute tfidf feature matrix
tf = np.array(bow_features, dtype='float64')
tfidf = tf * idf
# view raw tfidf feature matrix
pd.DataFrame(np.round(tfidf, 2), columns=feature_names)
```

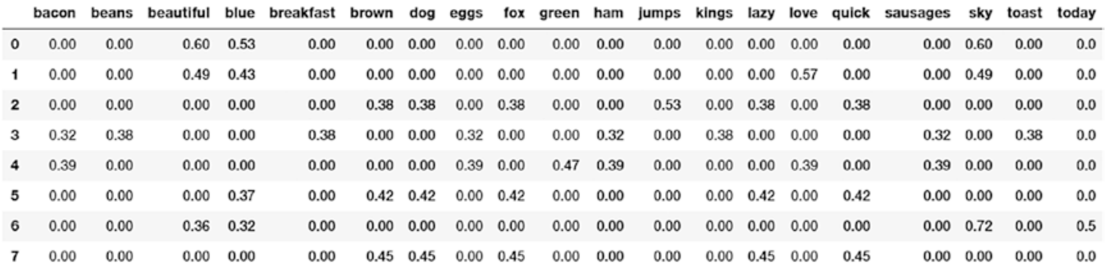


	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	1.81	0.0	0.0
1	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	2.1	0.00	0.0	1.81	0.0	0.0
2	0.0	0.0	0.00	0.00	0.0	1.81	1.81	0.0	1.81	0.0	0.0	2.5	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0
3	2.1	2.5	0.00	0.00	2.5	0.00	0.00	2.1	0.00	0.0	2.1	0.0	2.5	0.00	0.0	0.00	2.1	0.00	2.5	0.0
4	2.1	0.0	0.00	0.00	0.0	0.00	0.00	2.1	0.00	2.5	2.1	0.0	0.0	0.00	2.1	0.00	2.1	0.00	0.0	0.0
5	0.0	0.0	0.00	1.59	0.0	1.81	1.81	0.0	1.81	0.0	0.0	0.0	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0
6	0.0	0.0	1.81	1.59	0.0	0.00	0.00	0.0	0.00	0.0	0.0	0.0	0.0	0.00	0.0	0.00	0.0	3.62	0.0	2.5
7	0.0	0.0	0.00	0.00	0.0	1.81	1.81	0.0	1.81	0.0	0.0	0.0	0.0	1.81	0.0	1.81	0.0	0.00	0.0	0.0

Figure 4-10. *Constructing the raw TF-IDF matrix from the TF and IDF components*

We now have our `tfidf` feature matrix, but wait! We still have to divide this by the L2 norm, if you remember from our equations depicted earlier. The following snippet computes the `tfidf` norms for each document and then divides the `tfidf` weights by the norm to give us the final desired `tfidf` matrix. See Figure 4-11.

```
from numpy.linalg import norm
# compute L2 norms
norms = norm(tfidf, axis=1)

# print norms for each document
print (np.round(norms, 3))

[ 3.013  3.672  4.761  6.534  5.319  4.35   5.019  4.049]

# compute normalized tfidf
norm_tfidf = tfidf / norms[:, None]

# show final tfidf feature matrix
pd.DataFrame(np.round(norm_tfidf, 2), columns=feature_names)
```



	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.00	0.00	0.60	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.60	0.00	0.0
1	0.00	0.00	0.49	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.57	0.00	0.00	0.49	0.00	0.0
2	0.00	0.00	0.00	0.00	0.00	0.38	0.38	0.00	0.38	0.00	0.00	0.53	0.00	0.38	0.00	0.38	0.00	0.00	0.00	0.0
3	0.32	0.38	0.00	0.00	0.38	0.00	0.00	0.32	0.00	0.00	0.32	0.00	0.38	0.00	0.00	0.00	0.32	0.00	0.38	0.0
4	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.39	0.00	0.47	0.39	0.00	0.00	0.00	0.39	0.00	0.39	0.00	0.00	0.0
5	0.00	0.00	0.00	0.37	0.00	0.42	0.42	0.00	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.42	0.00	0.00	0.00	0.0
6	0.00	0.00	0.36	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.72	0.00	0.5
7	0.00	0.00	0.00	0.00	0.00	0.45	0.45	0.00	0.45	0.00	0.00	0.00	0.00	0.45	0.00	0.45	0.00	0.00	0.00	0.0

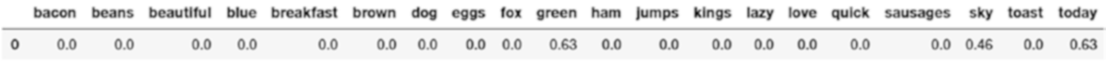
Figure 4-11. *Constructing the final normalized TF-IDF matrix*

If you compare obtained tfidf feature matrix in Figure 4-11 for the documents in our corpus to the feature matrix obtained using `TfidfTransformer` or `TfidfVectorizer` earlier. You will notice they are exactly the same, thus verifying that our mathematical implementation was correct. In fact, this very same implementation is adopted by Scikit-Learn behind the scenes using some more optimizations.

Extracting Features for New Documents

Suppose you built a machine learning model to classify and categorize news articles and it is in currently in production. How can you generate features for completely new documents so that you can feed it into the machine learning models for prediction? The Scikit-Learn API provides the `transform(...)` function for the vectorizers we discussed previously and we can leverage it to get features for a completely new document that was not present in our corpus (when we trained our model). See Figure 4-12.

```
new_doc = 'the sky is green today'
pd.DataFrame(np.round(tv.transform([new_doc]).toarray(), 2),
             columns=tv.get_feature_names())
```



	bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox	green	ham	jumps	kings	lazy	love	quick	sausages	sky	toast	today
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.63	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.46	0.0	0.63

Figure 4-12. *Generating the TF-IDF feature vector for a completely new document*

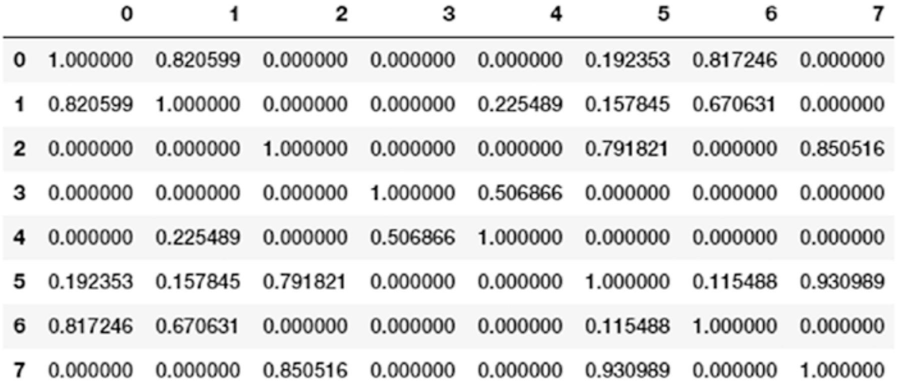
Thus, always leverage the `fit_transform(...)` function to build a feature matrix on all documents in your corpus. This typically becomes the training feature set on which you build and train your predictive or other machine learning models. Once ready, leverage the `transform(...)` function to generate feature vectors of new documents. This can then be fed into your trained models to generate insights as needed.

Document Similarity

Document similarity is the process of using a distance or similarity based metric that can identify how similar a text document is to any other document(s) based on features extracted from the documents, like Bag of Words or TF-IDF. Thus you can see that we can build on top of the TF-IDF-based features we engineered in the previous section and use them to generate new features. Domains such as search engines, document clustering, and information retrieval can be leveraged using these similarity based features.

Pairwise document similarity in a corpus involves computing document similarity for each pair of documents in a corpus. Thus, if you have **C** documents in a corpus, you would end up with a **C** x **C** matrix, such that each row and column represents the similarity score for a pair of documents. This represents the indices at the row and column, respectively. There are several similarity and distance metrics that are used to compute document similarity. These include cosine distance/similarity, Euclidean distance, manhattan distance, BM25 similarity, jaccard distance, and so on. In our analysis, we use perhaps the most popular and widely used similarity metrics—cosine similarity and compare pairwise document similarity—based on their TF-IDF feature vectors. See Figure 4-13.

```
from sklearn.metrics.pairwise import cosine_similarity

similarity_matrix = cosine_similarity(tv_matrix)
similarity_df = pd.DataFrame(similarity_matrix)
similarity_df
```



	0	1	2	3	4	5	6	7
0	1.000000	0.820599	0.000000	0.000000	0.000000	0.192353	0.817246	0.000000
1	0.820599	1.000000	0.000000	0.000000	0.225489	0.157845	0.670631	0.000000
2	0.000000	0.000000	1.000000	0.000000	0.000000	0.791821	0.000000	0.850516
3	0.000000	0.000000	0.000000	1.000000	0.506866	0.000000	0.000000	0.000000
4	0.000000	0.225489	0.000000	0.506866	1.000000	0.000000	0.000000	0.000000
5	0.192353	0.157845	0.791821	0.000000	0.000000	1.000000	0.115488	0.930989
6	0.817246	0.670631	0.000000	0.000000	0.000000	0.115488	1.000000	0.000000
7	0.000000	0.000000	0.850516	0.000000	0.000000	0.930989	0.000000	1.000000

Figure 4-13. *Pairwise document similarity matrix (cosine similarity)*

Cosine similarity gives us a metric representing the cosine of the angle between the feature vector representations of two text documents. The smaller the angle between the documents, the closer and more similar they are, as depicted with the scores in Figure 4-13 and with some sample document vectors in Figure 4-14.

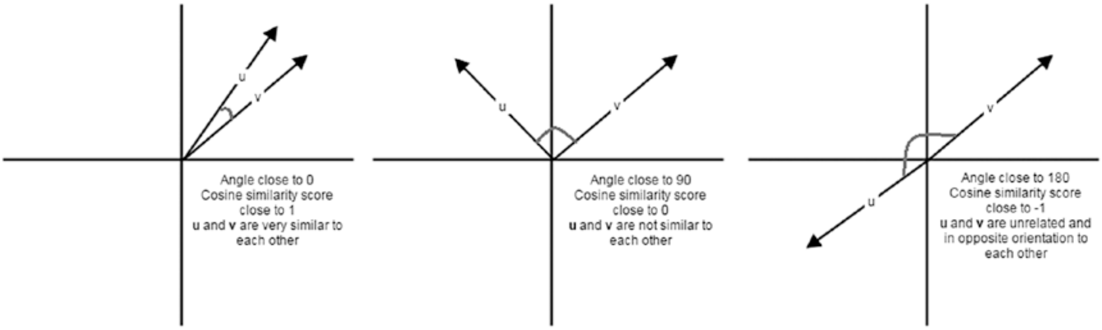


Figure 4-14. *Cosine similarity depictions for text document feature vectors*

Looking closely at the similarity matrix in Figure 4-13, you can clearly see that documents 0, 1, and 6 and 2, 5, and 7 are very similar to one another, whereas documents 3 and 4 are slightly similar to each other. This must indicate these similar documents have some similar features. This is a perfect example of grouping or clustering that can be solved by unsupervised learning, especially when you are dealing with huge corpora of millions of text documents.

Document Clustering with Similarity Features

We have been building a lot of features, but let's use some of them now for a real-world problem of grouping similar documents! Clustering leverages unsupervised learning to group data points (documents in this scenario) into groups or clusters. We leverage an unsupervised hierarchical clustering algorithm here to try to group similar documents from our toy corpus by leveraging the document similarity features we generated earlier.

There are two types of hierarchical clustering algorithms—agglomerative and divisive. We use an agglomerative clustering algorithm, which is hierarchical clustering using a bottom-up approach, i.e., each observation or document starts in its own cluster and clusters are successively merged using a distance metric that measures distances between data points and a linkage merge criterion. A sample depiction is shown in Figure 4-15.

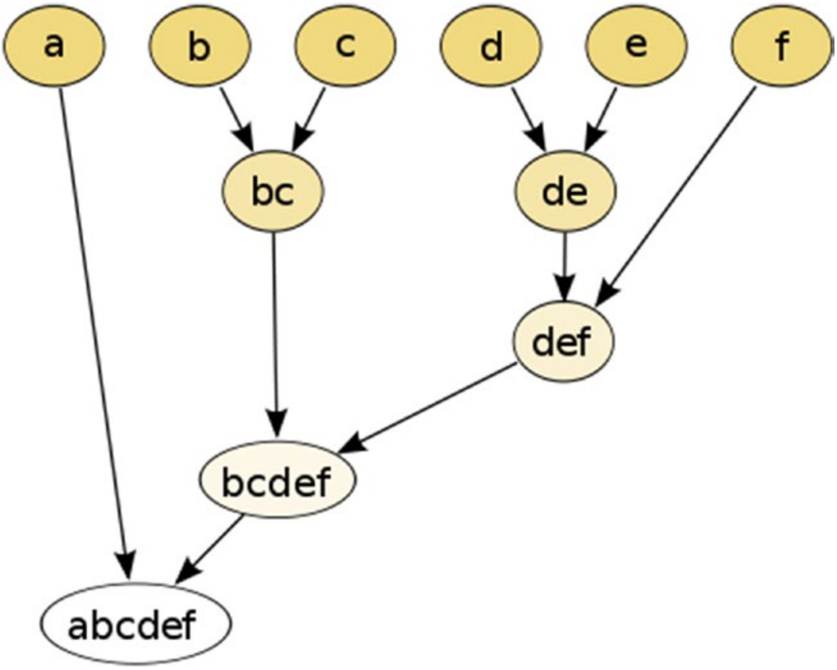
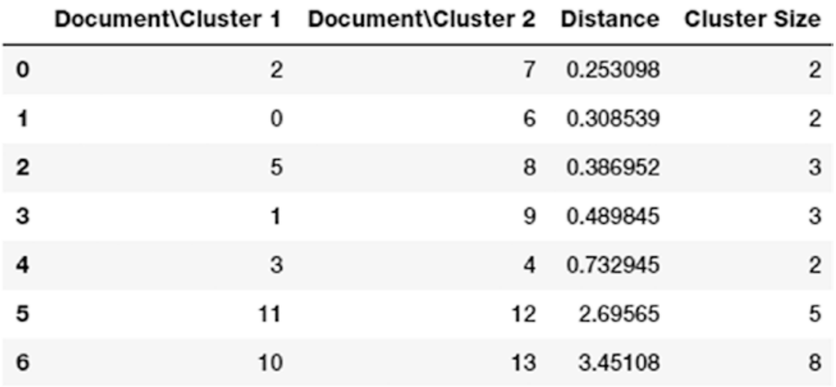


Figure 4-15. *Agglomerative hierarchical clustering*

The selection of the linkage criterion governs the merge strategy. Some examples of linkage criteria are Ward, Complete linkage, Average linkage, and so on. This criterion is very useful for choosing the pair of clusters (individual documents at the lowest step and clusters in higher steps) to merge at each step, which is based on the optimal value of an objective function. We choose the Ward's minimum variance method as our linkage criterion to minimize total within-cluster variance. Hence, at each step, we find the pair of clusters that leads to the minimum increase in total within-cluster variance after merging. Since we already have our similarity features, let's build the linkage matrix on our sample documents. See Figure 4-16.

```
from scipy.cluster.hierarchy import dendrogram, linkage

Z = linkage(similarity_matrix, 'ward')
pd.DataFrame(Z, columns=['Document\Cluster 1', 'Document\Cluster 2',
                         'Distance', 'Cluster Size'], dtype='object')
```



	Document\Cluster 1	Document\Cluster 2	Distance	Cluster Size
0	2	7	0.253098	2
1	0	6	0.308539	2
2	5	8	0.386952	3
3	1	9	0.489845	3
4	3	4	0.732945	2
5	11	12	2.69565	5
6	10	13	3.45108	8

Figure 4-16. *Linkage matrix for our corpus*

If you closely look at the linkage matrix in Figure 4-16, you can see that each step (row) of the linkage matrix tells us which data points (or clusters) were merged. If you have n data points, the linkage matrix, Z will have a shape of $(n - 1) \times 4$ where $Z[i]$ will tell us which clusters were merged at step i. Each row has four elements; the first two elements are either data point identifiers or cluster labels (in the later parts of the matrix once multiple data points are merged), the third element is the cluster distance between the first two elements (either data points or clusters), and the last element is the total number of elements/data points in the cluster once the merge is complete. We recommend you refer to the SciPy documentation, which explains this in detail. Let's now visualize this matrix as a dendrogram to understand the elements better! See Figure 4-17.

```
plt.figure(figsize=(8, 3))
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data point')
plt.ylabel('Distance')
dendrogram(Z)
plt.axhline(y=1.0, c='k', ls='--', lw=0.5)
```

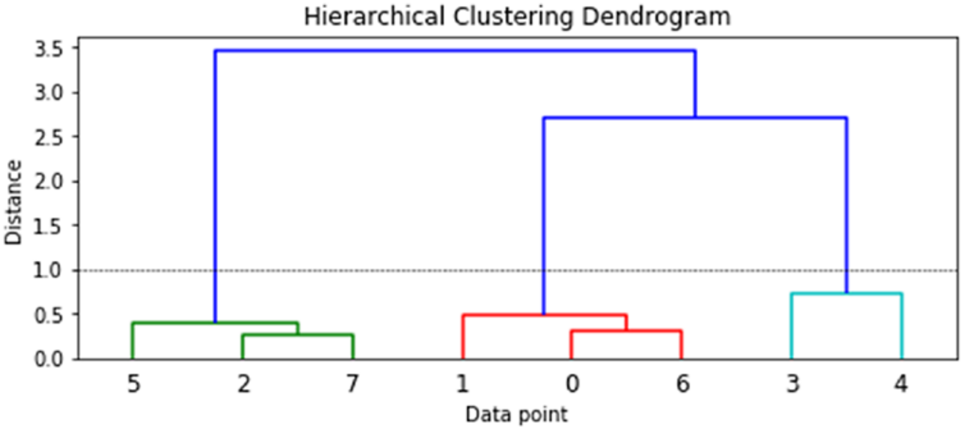
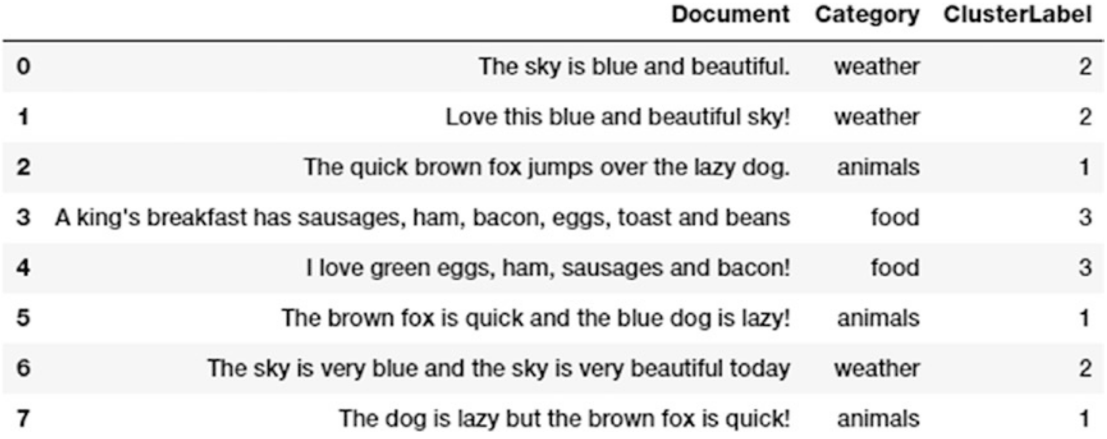


Figure 4-17. *Dendrogram visualizing our hierarchical clustering process*

We can see how each data point starts as an individual cluster and is slowly merged with other data points to form clusters. On a high level from the colors and the dendrogram, you can see that the model has correctly identified three major clusters if you consider a distance metric of around 1.0 or above (denoted by the dotted line). Leveraging this distance, we get our cluster labels. See Figure 4-18.

```
from scipy.cluster.hierarchy import fcluster
max_dist = 1.0

cluster_labels = fcluster(Z, max_dist, criterion='distance')
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```



	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	2
1	Love this blue and beautiful sky!	weather	2
2	The quick brown fox jumps over the lazy dog.	animals	1
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food	3
4	I love green eggs, ham, sausages and bacon!	food	3
5	The brown fox is quick and the blue dog is lazy!	animals	1
6	The sky is very blue and the sky is very beautiful today	weather	2
7	The dog is lazy but the brown fox is quick!	animals	1

Figure 4-18. *Clustering our documents into groups with hierarchical clustering*

Thus you can clearly see our algorithm has correctly identified the three distinct categories in our documents based on the cluster labels assigned to them. This should give you a good idea of how our TF-IDF features were leveraged to build our similarity features, which in turn helped in clustering our documents. You can use this pipeline in the future for clustering your own documents. We discuss text clustering in further detail with more models and examples in a future chapter in this book.

Topic Models

While we are covering topic modeling in detail in a separate chapter in this book, a discussion about feature engineering is not complete without talking about topic models. We can use some summarization techniques to extract topic- or concept-based features from text documents. The idea of topic models revolves around the process of extracting key themes or concepts from a corpus of documents, which are represented as topics. Each topic can be represented as a bag or collection of words/terms from the document corpus. Together, these terms signify a specific topic, theme, or a concept and each topic can be easily distinguished from other topics by virtue of the semantic meaning conveyed by these terms.

However, often you do end up with overlapping topics based on the data. These concepts can range from simple facts and statements to opinions and outlook. Topic models are extremely useful in summarizing large corpus of text documents to extract and depict key concepts. They are also useful in extracting features from text data that capture latent patterns in the data. See Figure 4-19.

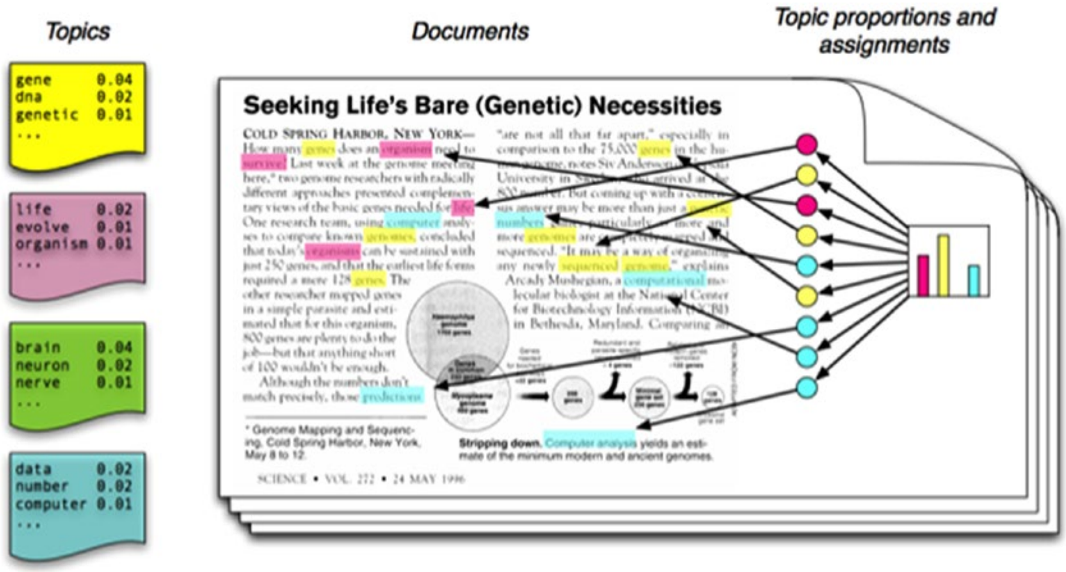


Figure source: Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, 55(4), 77-84.

***Figure 4-19.** Clustering our documents into groups using hierarchical clustering*

There are various techniques for topic modeling and most of them involve some form of matrix decomposition. Some techniques like Latent Semantic Indexing (LSI) use matrix decomposition operations, more specifically Singular Valued Decomposition. We use another technique called Latent Dirichlet Allocation (LDA), which uses a generative probabilistic model where each document consists of a combination of several topics and each term or word can be assigned to a specific topic. This is similar to the pLSI-based model (probabilistic LSI). Each latent topic contains a Dirichlet priority over them in the case of LDA. The math behind in this technique is pretty involved, so I will try to keep things simple here. We cover topic modeling in detail in a subsequent chapter also! See Figure 4-20.

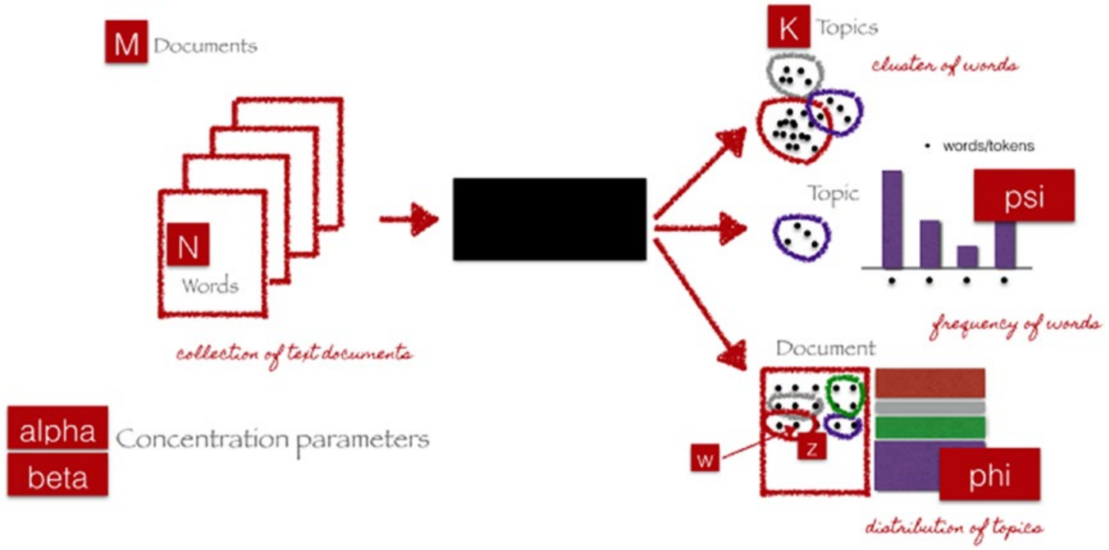
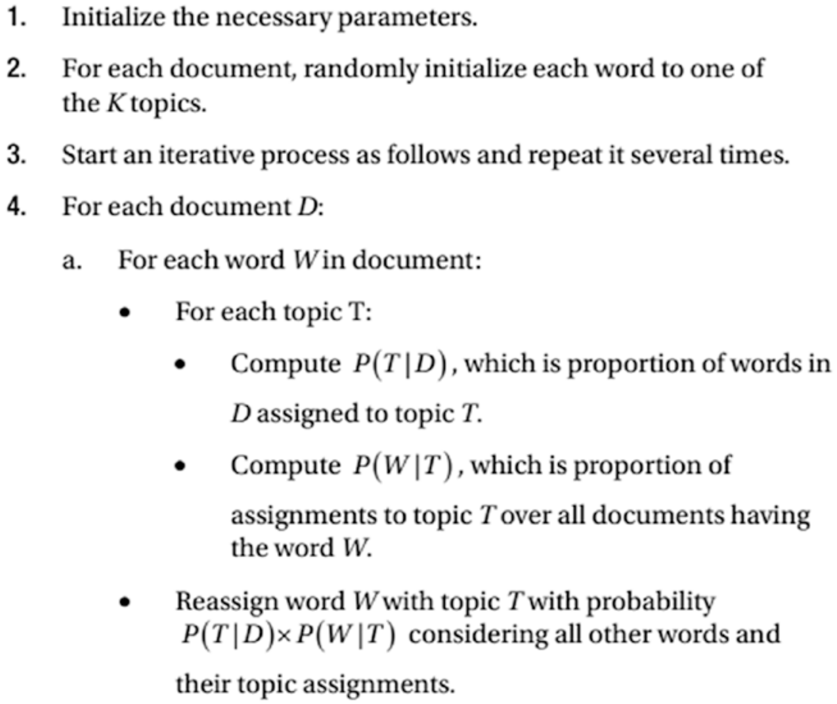


***Figure 4-20.** End-to-end LDA framework (courtesy of C. Doig, Introduction to Topic Modeling in Python)*

The black box in Figure 4-20 represents the core algorithm that uses the previously mentioned parameters to extract **K** topics from **M** documents. The steps outlined in Figure 4-22 give a simplistic explanation of what happens behind the scenes.



1. Initialize the necessary parameters.
2. For each document, randomly initialize each word to one of the *K* topics.
3. Start an iterative process as follows and repeat it several times.
4. For each document *D*:
    a. For each word *W* in document:
    - For each topic T:
        - Compute $P(T|D)$, which is proportion of words in *D* assigned to topic *T*.
        - Compute $P(W|T)$, which is proportion of assignments to topic *T* over all documents having the word *W*.
    - Reassign word *W* with topic *T* with probability $P(T|D) \times P(W|T)$ considering all other words and their topic assignments.

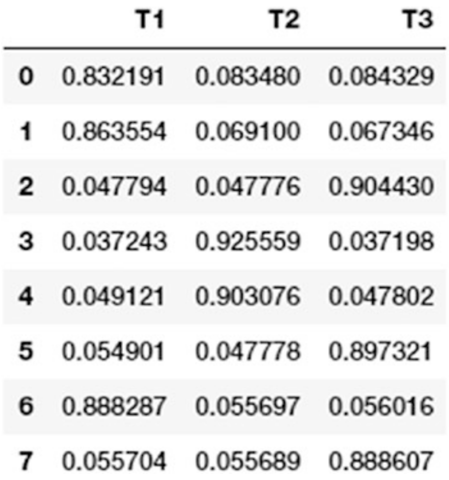
Figure 4-21. *Major steps in the LDA topic modeling algorithm*

Once this runs for several iterations, we should have topic mixtures for each document and then generate the constituents of each topic from the terms that point to that topic. Frameworks like Gensim or Scikit-Learn enable us to leverage the LDA model for generating topics. For the purpose of feature engineering, which is the intent of this chapter, you need to remember that when LDA is applied to a document-term matrix (TF-IDF or Bag of Words feature matrix), it is broken into two main components.

- A *document-topic matrix,* which would be the feature matrix we are looking for.
- A *topic-term matrix,* which helps us look at potential topics in the corpus.

Let's now leverage Scikit-Learn to get the document-topic matrix as follows. This can be used as features for any subsequent modeling requirements. See Figure 4-22.

```
from sklearn.decomposition import LatentDirichletAllocation

lda = LatentDirichletAllocation(n_topics=3, max_iter=10000, random_state=0)
dt_matrix = lda.fit_transform(cv_matrix)
features = pd.DataFrame(dt_matrix, columns=['T1', 'T2', 'T3'])
features
```



	T1	T2	T3
0	0.832191	0.083480	0.084329
1	0.863554	0.069100	0.067346
2	0.047794	0.047776	0.904430
3	0.037243	0.925559	0.037198
4	0.049121	0.903076	0.047802
5	0.054901	0.047778	0.897321
6	0.888287	0.055697	0.056016
7	0.055704	0.055689	0.888607

Figure 4-22. *Document-topic feature matrix from our LDA model*

You can clearly see which documents contribute the most to which of the three topics in this output. You can view the topics and their main constituents as follows.

```
tt_matrix = lda.components_
for topic_weights in tt_matrix:
    topic = [(token, weight) for token, weight in zip(vocab, topic_weights)]
    topic = sorted(topic, key=lambda x: -x[1])
    topic = [item for item in topic if item[1] > 0.6]
    print(topic)
    print()
```

```
[('sky', 4.3324395825632624), ('blue', 3.373753174831771), ('beautiful',
3.3323652405224857), ('today', 1.3325579841038182), ('love',
1.3304224288080069)]
```

```
[('bacon', 2.332695948479998), ('eggs', 2.332695948479998), ('ham',
2.332695948479998), ('sausages', 2.332695948479998), ('love',
1.335454457601996), ('beans', 1.332773525378464), ('breakfast',
1.332773525378464), ('kings', 1.332773525378464), ('toast',
1.332773525378464), ('green', 1.3325433207547732)]
```

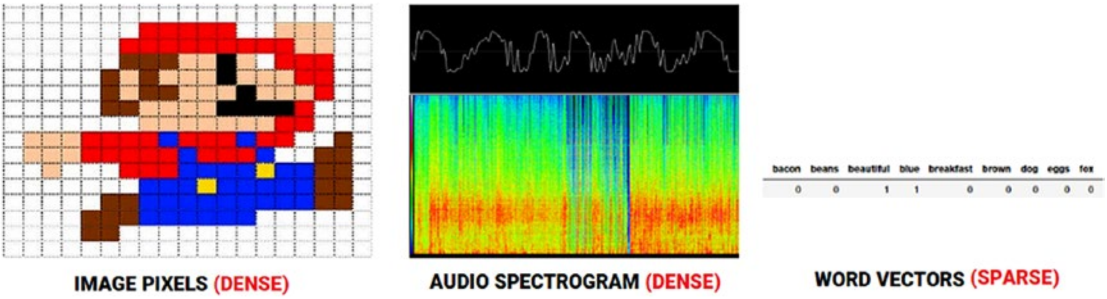
```
[('brown', 3.3323474595768783), ('dog', 3.3323474595768783), ('fox',
3.3323474595768783), ('lazy', 3.3323474595768783), ('quick',
3.3323474595768783), ('jumps', 1.3324193736202712), ('blue',
1.2919635624485213)]
```

Thus, you can clearly see the three topics are quite distinguishable from each other based on their constituent terms. The first one is talking about weather, the second one is about food, and the last one is about animals. Choosing the number of topics for topic modeling is an entire technique of its own and is an art as well as a science. There are various methods and heuristics to get the optimal number of topics, but due to the detailed nature of these techniques, we don't discuss them here.

Advanced Feature Engineering Models

Traditional (count-based) feature engineering strategies for textual data involve models belonging to a family of models, popularly known as the Bag of Words model. This includes term frequencies, TF-IDF (term frequency-inverse document frequency), N-Grams, and so on. While they are effective methods for extracting features from text, due to the inherent nature of the model being just a bag of unstructured words, we lose additional information like the semantics, structure, sequence, and context around nearby words in each text document. This forms as enough motivation for us to explore more sophisticated models that can capture this information and give us features that are vector representation of words, popularly known as *embeddings*.

While this does make some sense, why should we be motivated enough to learn and build these word embeddings? With regard to speech or image recognition systems, all the information is already present in the form of rich dense feature vectors embedded in high-dimensional datasets like audio spectrograms and image pixel intensities. However, when it comes to raw text data, especially count-based models like Bag of Words, we are dealing with individual words that may have their own identifiers and do not capture the semantic relationship among words. This leads to huge sparse word vectors for textual data and thus if we do not have enough data, we may end up getting poor models or even overfitting the data due to the curse of dimensionality. See Figure 4-23.



bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox
0	0	1	1	0	0	0	0	0

Figure 4-23. *Comparing feature representations for audio, image, and text*

To overcome the shortcomings of the Bag of Words model, we need to use *vector space models (VSMs)* in such a way that we can embed word vectors in this continuous vector space based on semantic and contextual similarity. In fact, the *distributional hypothesis* in the field of *distributional semantics* tells us that words that occur and are used in the same context are semantically similar to one another and have similar meanings. In simple terms, “a word is characterized by the company it keeps”.

One of the famous papers talking about these semantic word vectors and various types in detail is *“*Don’t count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors,” by Baroni et al. We won’t go into extensive depth, but in short, there are two main types of methods for contextual word vectors. *Count-based methods* like *Latent Semantic Analysis (LSA)* can be used to calculate statistical measures of how often words occur with their neighboring words in a corpus and then build dense word vectors for each word from these measures. *Predictive methods* like *neural network based language models* try to predict words from their neighboring words by looking at word sequences in the corpus. In the process, it learns distributed representations giving us dense word embeddings. We focus on these predictive methods in this section.

Loading the Bible Corpus

To train and showcase some of the capabilities of these advanced deep learning based feature representation models, we typically need a larger corpus. While we will still be using our previous corpus for demonstrations, let's also load our other corpus based on the King James version of the Bible using NLTK. Then we preprocess the text to showcase examples that might be more relevant depending on the models we implement later.

```
from nltk.corpus import gutenberg
from string import punctuation

bible = gutenberg.sents('bible-kjv.txt')
remove_terms = punctuation + '0123456789'

norm_bible = [[word.lower() for word in sent if word not in remove_terms]
              for sent in bible]
norm_bible = [' '.join(tok_sent) for tok_sent in norm_bible]
norm_bible = filter(None, normalize_corpus(norm_bible))
norm_bible = [tok_sent for tok_sent in norm_bible if len(tok_sent.split()) > 2]

print('Total lines:', len(bible))
print('\nSample line:', bible[10])
print('\nProcessed line:', norm_bible[10])
```

The following output shows the total number of lines in our corpus and how the preprocessing works on the Bible corpus.

```
Total lines: 30103

Sample line: ['1', ':', '6', 'And', 'God', 'said', ',', 'Let', 'there',
'be', 'a', 'firmament', 'in', 'the', 'midst', 'of', 'the', 'waters', ',',
'and', 'let', 'it', 'divide', 'the', 'waters', 'from', 'the', 'waters', '.']

Processed line: god said let firmament midst waters let divide waters
waters
```

Let's now look at some of the popular word embedding models and engineer meaningful features from our corpora!

Word2Vec Model

This model was created by Google in 2013 and is a predictive deep learning based model to compute and generate high quality, distributed, and continuous dense vector representations of words that capture contextual and semantic similarity. Essentially these are unsupervised models that can take in massive textual corpora, create a vocabulary of possible words, and generate dense word embeddings for each word in the vector space representing that vocabulary. Usually, you can specify the size of the word embedding vectors and the total number of vectors are essentially the size of the vocabulary. This makes the dimensionality of this dense vector space much lower than the high-dimensional sparse vector space built using traditional Bag of Words models.

There are two different model architectures that can be leveraged by Word2Vec to create these word embedding representations. These include:

- The Continuous Bag of Words (CBOW) model
- The Skip-Gram model

There were introduced by Mikolov et al. and I recommend interested readers read up on the original papers around these models, which includes "Distributed Representations of Words and Phrases and their Compositionality" by Mikolov et al. and "Efficient Estimation of Word Representations in Vector Space" by Mikolov et al. to gain an in-depth perspective.

The Continuous Bag of Words (CBOW) Model

The CBOW model (see Figure 4-24) architecture tries to predict the current target word (the center word) based on the source context words (surrounding words). Considering a simple sentence, "the quick brown fox jumps over the lazy dog", this can be pairs of (context _ window, target _ word), where if we consider a context window of size 2, we have examples like ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy), and so on. Thus, the model tries to predict the `target_word` based on the `context_window` words.

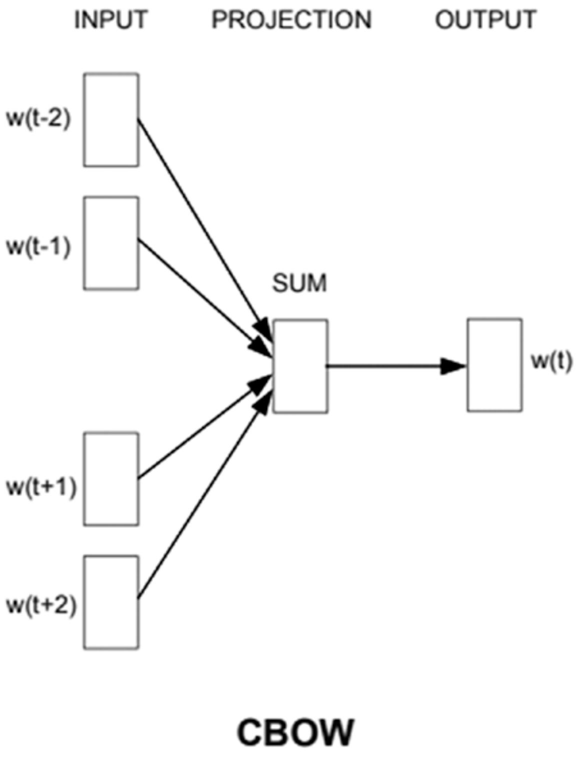


Figure 4-24. *The CBOW model architecture (Source: https://arxiv.org/pdf/1301.3781.pdf Mikolov et al.)*

Because the Word2Vec family of models is unsupervised, you can just give it a corpus without additional labels or information and it can construct dense word embeddings from the corpus. But you still need to leverage a supervised, classification methodology once you have this corpus to get to these embeddings. We do that from within the corpus itself, without any auxiliary information. We can model this CBOW architecture as a deep learning classification model such that we take in the context words as our input, **X**, and try to predict the target word, **Y**. In fact, building this architecture is simpler than the Skip-Gram model, whereby we try to predict a whole bunch of context words from a source target word.

Implementing the Continuous Bag of Words (CBOW) Model

While it's excellent to use robust frameworks that have the Word2Vec model like Gensim, let's try to implement this from scratch to gain some perspective on how things work behind the scenes. We leverage the Bible corpus contained in the `norm_bible` variable to train our model. The implementation will focus on four parts:

- Build the corpus vocabulary
- Build a CBOW (context, target) generator
- Build the CBOW model architecture
- Train the model
- Get word embeddings

Without further delay, let's get started!

Build the Corpus Vocabulary

To start off, we will build our corpus vocabulary, where we extract each unique word from our vocabulary and map a unique numeric identifier to it.

```
from keras.preprocessing import text
from keras.utils import np_utils
from keras.preprocessing import sequence

tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(norm_bible)
word2id = tokenizer.word_index

# build vocabulary of unique words
word2id['PAD'] = 0
id2word = {v:k for k, v in word2id.items()}
wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in
norm_bible]

vocab_size = len(word2id)
embed_size = 100
window_size = 2 # context window size
```

```
print('Vocabulary Size:', vocab_size)
print('Vocabulary Sample:', list(word2id.items())[:10])
```

```
Vocabulary Size: 12425
Vocabulary Sample: [('base', 2338), ('feller', 10771), ('sanctuary', 455),
('plunge', 10322), ('azariah', 1120), ('enlightened', 4438), ('horns',
838), ('kareah', 2920), ('nursing', 5943), ('baken', 3492)]
```

Thus, you can see that we created a vocabulary of unique words in our corpus and ways to map a word to its unique identifier and vice versa. The PAD term is typically used to pad context words to a fixed length if needed.

Build a CBOW (Context, Target) Generator

We need pairs that consist of a target center word and surround the context words. In our implementation, a target word is of length 1 and the surrounding context is of length $2 \times window\_size$, where we take window_size words before and after the target word in our corpus. This will become clearer with the following example.

```
def generate_context_word_pairs(corpus, window_size, vocab_size):
    context_length = window_size*2
    for words in corpus:
        sentence_length = len(words)
        for index, word in enumerate(words):
            context_words = []
            label_word   = []
            start = index - window_size
            end = index + window_size + 1

            context_words.append([words[i]
                                 for i in range(start, end)
                                 if 0 <= i < sentence_length
                                 and i != index])
            label_word.append(word)

            x = sequence.pad_sequences(context_words, maxlen=context_length)
            y = np_utils.to_categorical(label_word, vocab_size)
            yield (x, y)
```

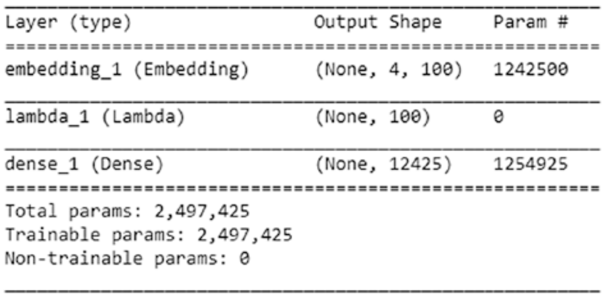
```
# Test this out for some samples
i = 0
for x, y in generate_context_word_pairs(corpus=wids, window_size=window_
size, vocab_size=vocab_size):
    if 0 not in x[0]:
        print('Context (X):', [id2word[w] for w in x[0]], '-> Target (Y):',
                  id2word[np.argwhere(y[0])[0][0]])
        if i == 10:
            break
        i += 1
Context (X): ['old','testament','james','bible'] -> Target (Y): king
Context (X): ['first','book','called','genesis'] -> Target(Y): moses
Context(X):['beginning','god','heaven','earth'] -> Target(Y):created
Context (X):['earth','without','void','darkness'] -> Target(Y): form
Context (X): ['without','form','darkness','upon'] -> Target(Y): void
Context (X): ['form', 'void', 'upon', 'face'] -> Target(Y): darkness
Context (X): ['void', 'darkness', 'face', 'deep'] -> Target(Y): upon
Context (X): ['spirit', 'god', 'upon', 'face'] -> Target (Y): moved
Context (X): ['god', 'moved', 'face', 'waters'] -> Target (Y): upon
Context (X): ['god', 'said', 'light', 'light'] -> Target (Y): let
Context (X): ['god', 'saw', 'good', 'god'] -> Target (Y): light
```

The preceding output should give you some more perspective of how **X** forms our context words and we are trying to predict the target center word **Y**, based on this context. For example, say the original text was “in the beginning god created heaven and earth” which, after preprocessing and removal of stopwords, became “beginning god created heaven earth”. Given [beginning, god, heaven, earth] as the context, the target center word is “created” in this case.

Build the CBOW Model Architecture

We now leverage Keras on top of TensorFlow to build our deep learning architecture for the CBOW model. For this, our inputs will be our context words, which are passed to an embedding layer (initialized with random weights). The word embeddings are propagated to a lambda layer where we average the word embeddings (hence called CBOW because we don't really consider the order or sequence in the context words when averaged). Then we pass this averaged context embedding to a dense softmax layer, which predicts our target word. We match this with the actual target word, compute the loss by leveraging the `categorical_crossentropy` loss, and perform back-propagation with each epoch to update the embedding layer in the process. He following code shows the model architecture. See Figure 4-25.

```
import keras.backend as K
from keras.models import Sequential
from keras.layers import Dense, Embedding, Lambda

# build CBOW architecture
cbow = Sequential()
cbow.add(Embedding(input_dim=vocab_size, output_dim=embed_size, input_
length=window_size*2))
cbow.add(Lambda(lambda x: K.mean(x, axis=1), output_shape=(embed_size,)))
cbow.add(Dense(vocab_size, activation='softmax'))
cbow.compile(loss='categorical_crossentropy', optimizer='rmsprop')

# view model summary
print(cbow.summary())

# visualize model structure
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(cbow, show_shapes=True, show_layer_names=False,
                 rankdir='TB').create(prog='dot', format='svg'))
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 4, 100)            1242500   
_________________________________________________________________
lambda_1 (Lambda)            (None, 100)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 12425)             1254925   
=================================================================
Total params: 2,497,425
Trainable params: 2,497,425
Non-trainable params: 0
_________________________________________________________________
```

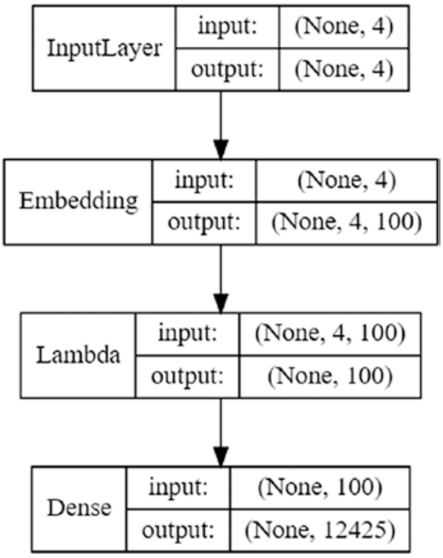


Figure 4-25. *CBOW model summary and architecture*

In case you still have difficulty visualizing this deep learning model, I recommend you read through the papers I mentioned earlier. I try to summarize the core concepts of this model in simple terms. We have input context words of dimensions (2 × *window _ size*), and we will pass them to an embedding layer of size (*vocab _ size* × *embed _ size*), which will give us dense word embeddings for each of these context words (1 × *embed _ size for each word*). Next, we use a lambda layer to average these embeddings and get an average dense embedding (1 × *embed _ size*), which is sent to the dense softmax layer, which outputs the most likely target word. We compare this with the actual target word, compute the loss, back-propagate the errors to adjust the weights (in the embedding layer), and repeat this process for all (context, target) pairs for multiple epochs. Figure 4-26 tries to explain this process.

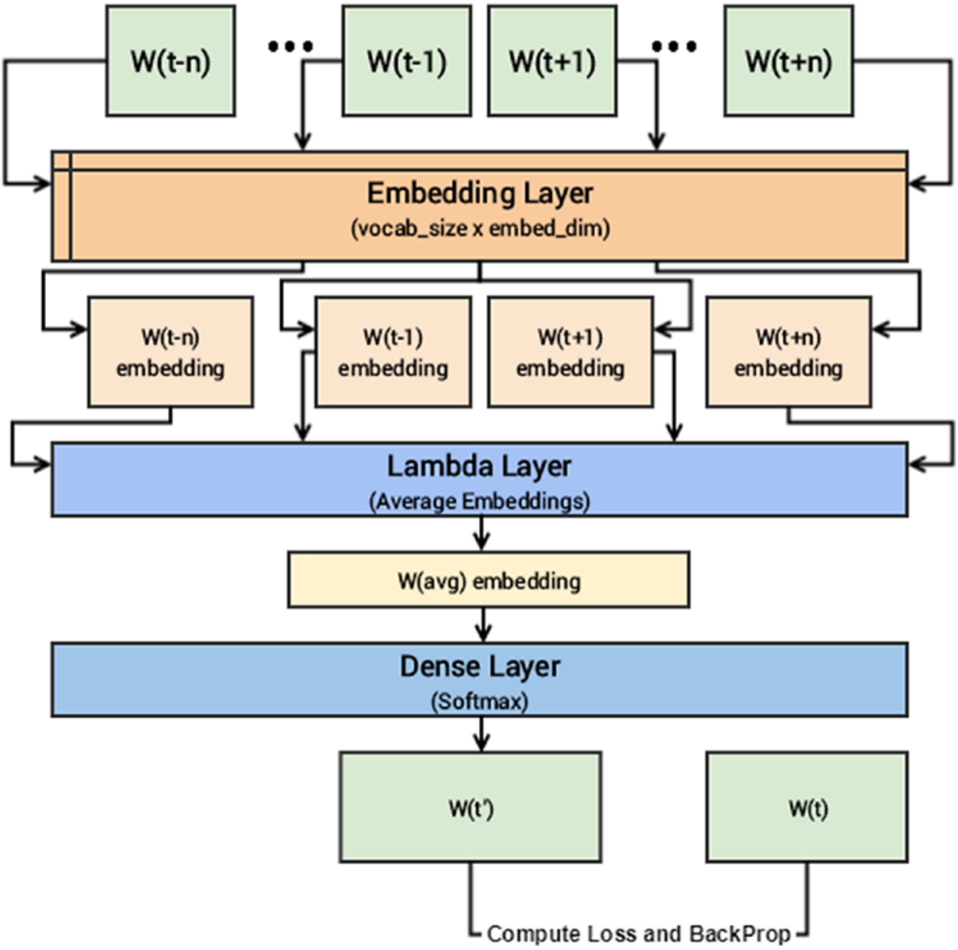


Figure 4-26. *Visual depiction of the CBOW deep learning model*

We are now ready to train this model on our corpus using our data generator to feed in the (context, target_word) pairs.

Train the Model

Running the model on our complete corpus takes a fair bit of time, so I just ran it for five epochs. You can leverage the following code and increase it for more epochs if necessary.

```
for epoch in range(1, 6):
    loss = 0.
    i = 0
    for x, y in generate_context_word_pairs(corpus=wids, window_
    size=window_size, vocab_size=vocab_size):

        i += 1
        loss += cbow.train_on_batch(x, y)
        if i % 100000 == 0:
            print('Processed {} (context, word) pairs'.format(i))

    print('Epoch:', epoch, '\tLoss:', loss)
    print()
```

```
Epoch: 1     Loss: 4257900.60084
Epoch: 2     Loss: 4256209.59646
Epoch: 3     Loss: 4247990.90456
Epoch: 4     Loss: 4225663.18927
Epoch: 5     Loss: 4104501.48929
```

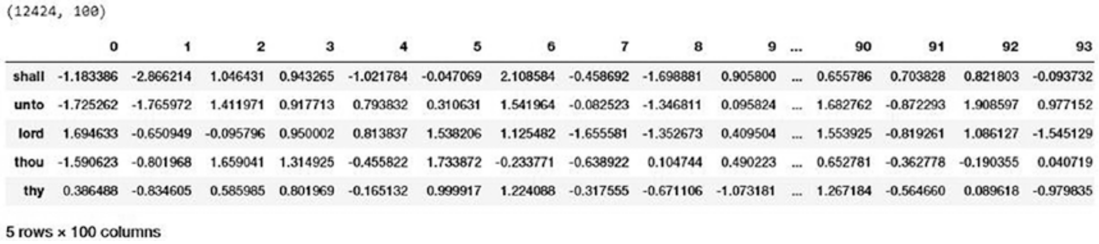
Note Running this model is computationally expensive and works better if trained using a GPU. I trained this on an AWS p2.x instance with a Tesla K80 GPU and it took me close to 1.5 hours for just five epochs!

Once this model is trained, similar words should have similar weights based on the embedding layer.

Get Word Embeddings

To get word embeddings for our entire vocabulary, we can extract them from our embedding layer by leveraging the following code. We don't take the embedding at position 0 since it belongs to the padding (PAD) term, which is not really a word of interest. See Figure 4-27.

```
weights = cbow.get_weights()[0]
weights = weights[1:]
print(weights.shape)

pd.DataFrame(weights, index=list(id2word.values())[1:]).head()
```



```
(12424, 100)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93
shall	-1.183386	-2.866214	1.046431	0.943265	-1.021784	-0.047069	2.108584	-0.458692	-1.698881	0.905800	...	0.655786	0.703828	0.821803	-0.093732
unto	-1.725262	-1.765972	1.411971	0.917713	0.793832	0.310631	1.541964	-0.082523	-1.346811	0.095824	...	1.682762	-0.872293	1.908597	0.977152
lord	1.694633	-0.650949	-0.095796	0.950002	0.813837	1.538206	1.125482	-1.655581	-1.352673	0.409504	...	1.553925	-0.819261	1.086127	-1.545129
thou	-1.590623	-0.801968	1.659041	1.314925	-0.455822	1.733872	-0.233771	-0.638922	0.104744	0.490223	...	0.652781	-0.362778	-0.190355	0.040719
thy	0.386488	-0.834605	0.585985	0.801969	-0.165132	0.999917	1.224088	-0.317555	-0.671106	-1.073181	...	1.267184	-0.564660	0.089618	-0.979835

5 rows × 100 columns

Figure 4-27. *Word embeddings for our vocabulary based on the CBOW model*

Thus, you can clearly see that each word has a dense embedding of size (1 × 100), as depicted in the output in Figure 4-27. Let's try to find some contextually similar words for specific words of interest based on these embeddings. For this, we build a pairwise distance matrix among all the words in our vocabulary based on the dense embedding vectors and then find the n-nearest neighbors of each word of interest based on the shortest (Euclidean) distance.

```
from sklearn.metrics.pairwise import euclidean_distances

# compute pairwise distance matrix
distance_matrix = euclidean_distances(weights)
print(distance_matrix.shape)

# view contextually similar words
similar_words = {search_term: [id2word[idx]
                   for idx in distance_matrix[word2id[search_term]-1].
                   argsort()[1:6]+1]
                      for search_term in ['god', 'jesus', 'noah', 'egypt',
                      'john', 'gospel', 'moses','famine']}
similar_words

(12424, 12424)
{'egypt': ['destroy', 'none', 'whole', 'jacob', 'sea'],
 'famine': ['wickedness', 'sore', 'countries', 'cease', 'portion'],
 'god': ['therefore', 'heard', 'may', 'behold', 'heaven'],
 'gospel': ['church', 'fowls', 'churches', 'preached', 'doctrine'],
 'jesus': ['law', 'heard', 'world', 'many', 'dead'],
 'john': ['dream', 'bones', 'held', 'present', 'alive'],
 'moses': ['pharaoh', 'gate', 'jews', 'departed', 'lifted'],
 'noah': ['abram', 'plagues', 'hananiah', 'korah', 'sarah']}
```

You can clearly see that some of these make sense contextually (god, heaven), (gospel, church) and so on and some do not. Training for more epochs usually ends up giving better results. We now explore the Skip-Gram architecture, which often gives better results than CBOW.

The Skip-Gram Model

The Skip-Gram model architecture tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the center word). Consider our simple sentence from earlier, "the quick brown fox jumps over the lazy dog". If we used the CBOW model, we get pairs of (context_window, target_word), where if we consider a context window of size 2, we have examples such as ([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy) and so on. Now, considering that the Skip-Gram model's aim is to predict the context from the target word, the model typically inverts the contexts and targets and tries to predict each context word from its target word. Hence the task becomes to predict the context [quick, fox], given target word "brown" or [the, brown] given target word "quick," and so on. Thus the model tries to predict the `context_window` words based on the `target_word`. See Figure 4-28.

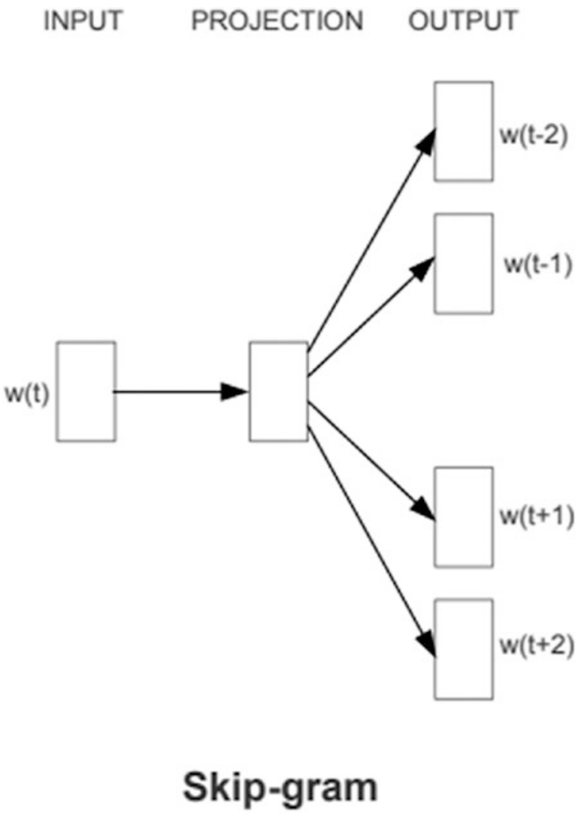


Figure 4-28. *The Skip-Gram model architecture (Source: https://arxiv.org/pdf/1301.3781.pdf Mikolov et al.)*

As we discussed in the CBOW model, we need to model this Skip-Gram architecture as a deep learning classification model, so we take in the target word as our input and try to predict the context words. This becomes slightly complex since we have multiple words in our context. We simplify this further by breaking down each $(target, context\_words)$ pair into $(target, context)$ pairs so the context consists of only one word. Hence our dataset from earlier gets transformed into pairs like (brown, quick), (brown, fox), (quick, the), (quick, brown), and so on. But how do we supervise or train the model to know what is contextual and what is not?

For this, we feed our Skip-Gram model pairs of (X, Y), where X is our input and Y is our label. We do this by using $[(target, context), 1]$ pairs as positive input samples, where *target* is our word of interest and *context* is a context word occurring near the target word. The *positive label 1* indicates this is a contextually relevant pair. We also feed in $[(target, random), 0]$ pairs as negative input samples, where *target* is again our word of interest but *random* is just a randomly selected word from our vocabulary and it has no context or association with our target word. Hence the *negative label 0* indicates this is a contextually irrelevant pair. We do this so that the model can then learn which pairs of words are contextually relevant and which are not and then generate similar embeddings for semantically similar words.

Implementing the Skip-Gram Model

Let's now try to implement this model from scratch to gain some perspective on how things work behind the scenes and so that we can compare it to our implementation of the CBOW model. We leverage our Bible corpus as usual, which is contained in the `norm_bible` variable for training our model. The implementation will focus on five parts:

- Build the corpus vocabulary
- Build a Skip-Gram [(target, context), relevancy] generator
- Build the Skip-Gram model architecture
- Train the model
- Get word embeddings

Let's get cracking and build our Skip-Gram Word2Vec model!

Build the Corpus Vocabulary

To start, we follow the standard process of building our corpus vocabulary where we extract each unique word from our vocabulary and assign a unique identifier, similar to what we did in the CBOW model. We also maintain mappings to transform words to their unique identifiers and vice versa.

```
from keras.preprocessing import text

tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(norm_bible)

word2id = tokenizer.word_index
id2word = {v:k for k, v in word2id.items()}

vocab_size = len(word2id) + 1
embed_size = 100
```

```
wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in
norm_bible]
print('Vocabulary Size:', vocab_size)
print('Vocabulary Sample:', list(word2id.items())[:10])
```

```
Vocabulary Size: 12425
Vocabulary Sample: [('base', 2338), ('feller', 10771), ('sanctuary', 455),
('plunge', 10322), ('azariah', 1120), ('enlightened', 4438), ('horns',
838), ('kareah', 2920), ('nursing', 5943), ('baken', 3492)]
```

Each unique word from the corpus is now part of our vocabulary and has a unique numeric identifier.

Build a Skip-Gram [(target, context), relevancy] Generator

It's now time to build our Skip-Gram generator, which will give us pair of words and their relevance, as we discussed earlier. Luckily, Keras has a nifty Skip-Grams utility that can be used and we don't have to manually implement this generator like we did in CBOW.

> The function `skipgrams(...)` is present in `keras.preprocessing.sequence`. This function transforms a sequence of word indexes (list of integers) into tuples of words of the form:
>
> 1. (word, word in the same window), with label 1 (positive samples).
> 2. (word, random word from the vocabulary), with label 0 (negative samples).

```
from keras.preprocessing.sequence import skipgrams

# generate skip-grams
skip_grams = [skipgrams(wid, vocabulary_size=vocab_size, window_size=10)
for wid in wids]

# view sample skip-grams
pairs, labels = skip_grams[0][0], skip_grams[0][1]
for i in range(10):
    print("({:s} ({:d}), {:s} ({:d})) -> {:d}".format(
          id2word[pairs[i][0]], pairs[i][0],
          id2word[pairs[i][1]], pairs[i][1],
          labels[i]))
```

```
(bible (5766), stank (5220)) -> 0
(james (1154), izri (9970)) -> 0
(king (13), bad (2285)) -> 0
(king (13), james (1154)) -> 1
(king (13), lucius (8272)) -> 0
(james (1154), king (13)) -> 1
(james (1154), bazluth (10091)) -> 0
(james (1154), bible (5766)) -> 1
(king (13), bible (5766)) -> 1
(bible (5766), james (1154)) -> 1
```

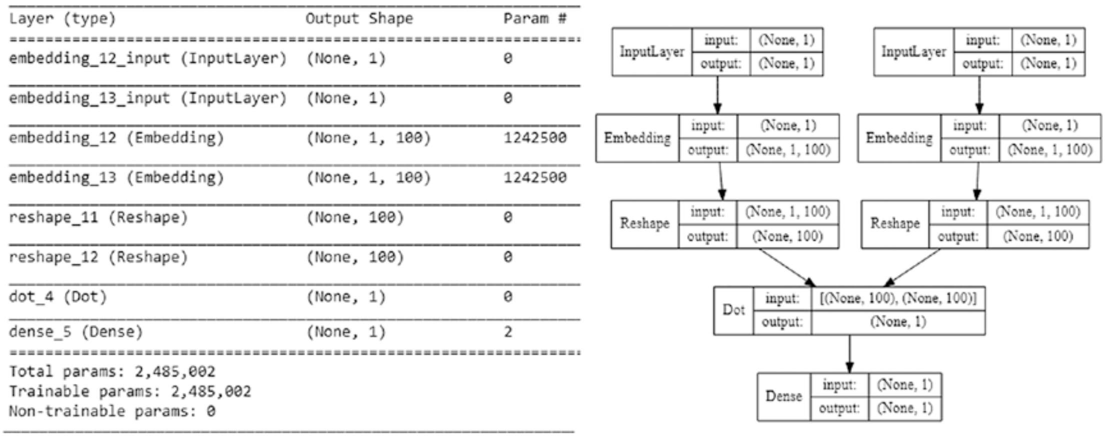
Thus, you can see we have successfully generated our required Skip-Grams and, based on the sample Skip-Grams in the preceding output, you can clearly see what is relevant and what is irrelevant based on the label (0 or 1).

Build the Skip-Gram Model Architecture

We now leverage Keras on top of TensorFlow to build our deep learning architecture for the Skip-Gram model. For this, our inputs will be our target word and context or random word pair. Each of these are passed to an embedding layer (initialized with random weights) of its own. Once we obtain the word embeddings for the target and the context word, we pass it to a merge layer where we compute the dot product of these two vectors. Then we pass this dot product value to a dense sigmoid layer, which predicts either a 1 or a 0 depending on if the pair of words are contextually relevant or just random words (Y').

We match this with the actual relevance label (Y), compute the loss by leveraging the `mean_squared_error` loss, and perform back-propagation with each epoch to update the embedding layer in the process. The following code shows our model architecture. See Figure 4-29.

```
from keras.layers import Dot
from keras.layers.core import Dense, Reshape
from keras.layers.embeddings import Embedding
from keras.models import Sequential
from keras.models import Model

# build skip-gram architecture
word_model = Sequential()
word_model.add(Embedding(vocab_size, embed_size, embeddings_initializer=
                         "glorot_uniform", input_length=1))
word_model.add(Reshape((embed_size, )))

context_model = Sequential()
context_model.add(Embedding(vocab_size, embed_size,
                  embeddings_initializer="glorot_uniform",
                  input_length=1))
context_model.add(Reshape((embed_size,)))
model_arch = Dot(axes=1)([word_model.output, context_model.output])
model_arch = Dense(1, kernel_initializer="glorot_uniform",
activation="sigmoid")(model_arch)
model = Model([word_model.input,context_model.input], model_arch)
model.compile(loss="mean_squared_error", optimizer="rmsprop")

# view model summary
print(model.summary())

# visualize model structure
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True, show_layer_names=False,
                 rankdir='TB').create(prog='dot', format='svg'))
```



```
Layer (type)                     Output Shape          Param #
================================================================
embedding_12_input (InputLayer)  (None, 1)             0
________________________________________________________________
embedding_13_input (InputLayer)  (None, 1)             0
________________________________________________________________
embedding_12 (Embedding)         (None, 1, 100)        1242500
________________________________________________________________
embedding_13 (Embedding)         (None, 1, 100)        1242500
________________________________________________________________
reshape_11 (Reshape)             (None, 100)           0
________________________________________________________________
reshape_12 (Reshape)             (None, 100)           0
________________________________________________________________
dot_4 (Dot)                      (None, 1)             0
________________________________________________________________
dense_5 (Dense)                  (None, 1)             2
================================================================
Total params: 2,485,002
Trainable params: 2,485,002
Non-trainable params: 0
________________________________________________________________
```

InputLayer | input: (None, 1) | output: (None, 1)
InputLayer | input: (None, 1) | output: (None, 1)
Embedding | input: (None, 1) | output: (None, 1, 100)
Embedding | input: (None, 1) | output: (None, 1, 100)
Reshape | input: (None, 1, 100) | output: (None, 100)
Reshape | input: (None, 1, 100) | output: (None, 100)
Dot | input: [(None, 100), (None, 100)] | output: (None, 1)
Dense | input: (None, 1) | output: (None, 1)

Figure 4-29. *Skip-Gram model summary and architecture*

Understanding the deep learning model is pretty straightforward. However, I will try to summarize the core concepts of this model in simple terms for ease of understanding. We have a pair of input words for each training example consisting of one input target word with a unique numeric identifier and one context word with a unique numeric identifier. If it is a positive sample, the word has contextual meaning, is a context word, and our label $Y = 1$. Otherwise, if it is a negative sample, the word has no contextual meaning, is just a random word, and our label $Y = 0$. We will pass each of these to an embedding layer of their own, having size ($\boldsymbol{vocab\_size \times embed\_size}$), which will give us dense word embeddings for each of these two words ($\boldsymbol{1 \times embed\_size\, for\, each\, word}$).

Next, we use a merge layer to compute the dot product of these two embeddings and get the dot product value. This is then sent to the dense sigmoid layer, which outputs a 1 or 0. We compare this to the actual label $Y\,(1\ or\ 0)$, compute the loss, back-propagate the errors to adjust the weights (in the embedding layer), and repeat this process for all $(target, context)$ pairs for multiple epochs. Figure 4-30 tries to explain this process.

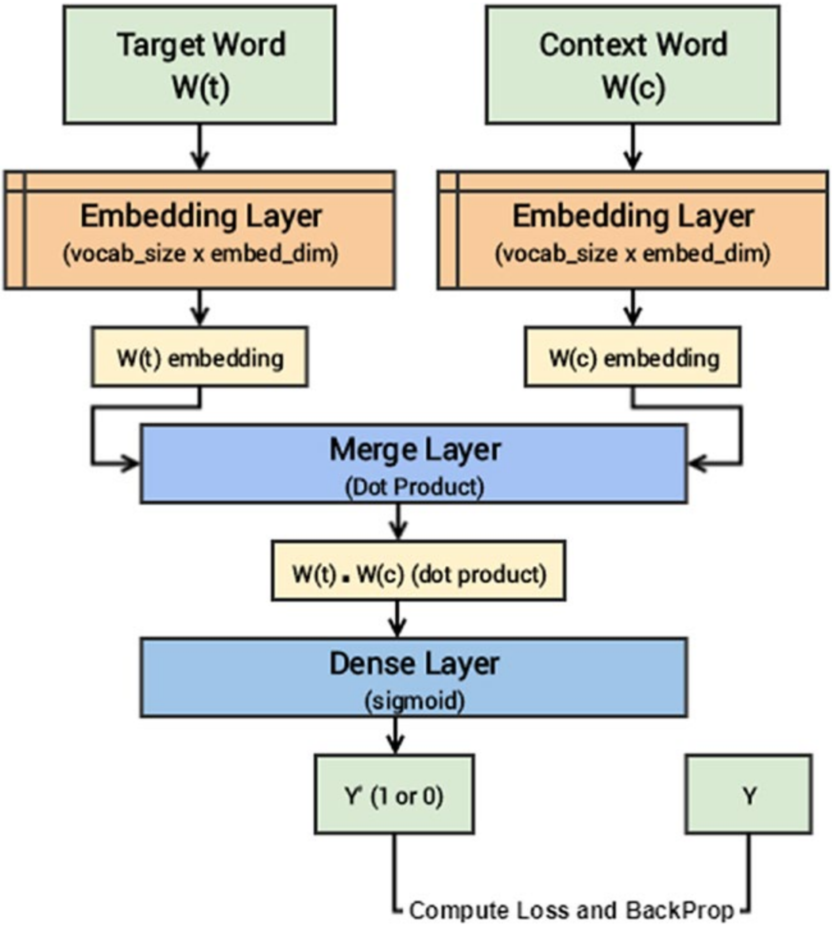


Figure 4-30. *Visual depiction of the Skip-Gram deep learning model*

Let's now start training our model with our Skip-Grams.

Train the Model

Running the model on our complete corpus takes a fair bit of time, but it's quicker than the CBOW model. I ran it for five epochs. You can leverage the following code and run more epochs if necessary.

```
for epoch in range(1, 6):
    loss = 0
    for i, elem in enumerate(skip_grams):
        pair_first_elem = np.array(list(zip(*elem[0]))[0], dtype='int32')
        pair_second_elem = np.array(list(zip(*elem[0]))[1], dtype='int32')
        labels = np.array(elem[1], dtype='int32')
        X = [pair_first_elem, pair_second_elem]
        Y = labels
        if i % 10000 == 0:
            print('Processed {} (skip_first, skip_second, relevance)
            pairs'.format(i))
        loss += model.train_on_batch(X,Y)

    print('Epoch:', epoch, 'Loss:', loss)
```

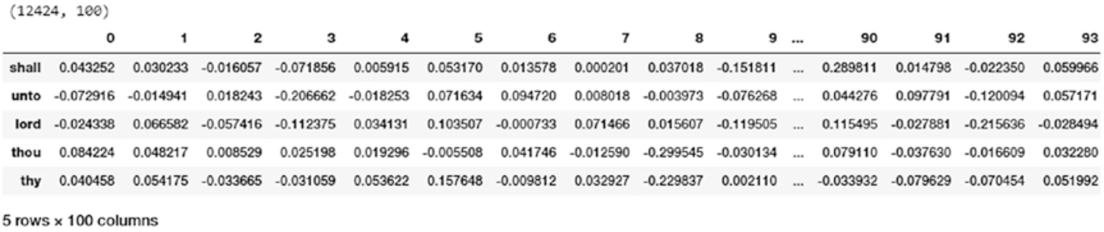
```
Epoch: 1 Loss: 4474.41281086
Epoch: 2 Loss: 3750.71884749
Epoch: 3 Loss: 3752.47489296
Epoch: 4 Loss: 3793.9177565
Epoch: 5 Loss: 3718.15081862
```

Once this model is trained, similar words should have similar weights based on the embedding layer.

Get Word Embeddings

To get word embeddings for our entire vocabulary, we can extract them from our embedding layer by leveraging the following code. Note that we are only interested in the target word embedding layer, so we extract the embeddings from our `word_model` embedding layer. We don't take the embedding at position 0 since none of our words in the vocabulary have a numeric identifier of 0. See Figure 4-31.

```
word_embed_layer = model.layers[2]
weights = word_embed_layer.get_weights()[0][1:]

print(weights.shape)
pd.DataFrame(weights, index=id2word.values()).head()
```



```
(12424, 100)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93
shall	0.043252	0.030233	-0.016057	-0.071856	0.005915	0.053170	0.013578	0.000201	0.037018	-0.151811	...	0.289811	0.014798	-0.022350	0.059966
unto	-0.072916	-0.014941	0.018243	-0.206662	-0.018253	0.071634	0.094720	0.008018	-0.003973	-0.076268	...	0.044276	0.097791	-0.120094	0.057171
lord	-0.024338	0.066582	-0.057416	-0.112375	0.034131	0.103507	-0.000733	0.071466	0.015607	-0.119505	...	0.115495	-0.027881	-0.215636	-0.028494
thou	0.084224	0.048217	0.008529	0.025198	0.019296	-0.005508	0.041746	-0.012590	-0.299545	-0.030134	...	0.079110	-0.037630	-0.016609	0.032280
thy	0.040458	0.054175	-0.033665	-0.031059	0.053622	0.157648	-0.009812	0.032927	-0.229837	0.002110	...	-0.033932	-0.079629	-0.070454	0.051992

5 rows × 100 columns

Figure 4-31. *Word embeddings for our vocabulary based on the Skip-Gram model*

Thus, you can clearly see that each word has a dense embedding of size (1 × 100), as depicted in the preceding output. This is similar to what we obtained from the CBOW model. Let's now apply the Euclidean distance metric on these dense embedding vectors to generate a pairwise distance metric for each word in our vocabulary. We can then determine the n-nearest neighbors of each word of interest based on the shortest (Euclidean) distance, similar to what we did on the embeddings from our CBOW model.

```
from sklearn.metrics.pairwise import euclidean_distances

distance_matrix = euclidean_distances(weights)
print(distance_matrix.shape)

similar_words = {search_term: [id2word[idx]
                       for idx in distance_matrix[word2id[search_term]-1].
                       argsort()[1:6]+1]
                          for search_term in ['god', 'jesus', 'noah',
                          'egypt', 'john', 'gospel', 'moses','famine']}
similar_words
```

```
(12424, 12424)
{'egypt': ['taken', 'pharaoh', 'wilderness', 'gods', 'became'],
 'famine': ['moved', 'awake', 'driven', 'howl', 'snare'],
 'god': ['strength', 'given', 'blessed', 'wherefore', 'lord'],
 'gospel': ['preached', 'must', 'preach', 'desire', 'grace'],
 'jesus': ['disciples', 'christ', 'dead', 'peter', 'jews'],
 'john': ['peter', 'hold', 'mountain', 'ghost', 'preached'],
 'moses': ['commanded', 'third', 'congregation', 'tabernacle', 'tribes'],
 'noah': ['ham', 'terah', 'amon', 'adin', 'zelophehad']}
```

You can clearly see from the results that a lot of the similar words for each of the words of interest are making sense and we have obtained better results as compared to our CBOW model. Let's visualize these word embeddings using t-SNE, which stands for *t-distributed stochastic neighbor embedding*. It's a popular dimensionality reduction technique used to visualize higher dimension spaces in lower dimensions (e.g. 2D). See Figure 4-32.

```
from sklearn.manifold import TSNE

words = sum([[k] + v for k, v in similar_words.items()], [])
words_ids = [word2id[w] for w in words]
word_vectors = np.array([weights[idx] for idx in words_ids])
print('Total words:', len(words), '\tWord Embedding shapes:', word_vectors.
shape)

tsne = TSNE(n_components=2, random_state=0, n_iter=10000, perplexity=3)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(word_vectors)
labels = words

plt.figure(figsize=(14, 8))
plt.scatter(T[:, 0], T[:, 1], c='steelblue', edgecolors='k')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset
    points')
```

```
Total words: 48          Word Embedding shapes: (48, 100)
```

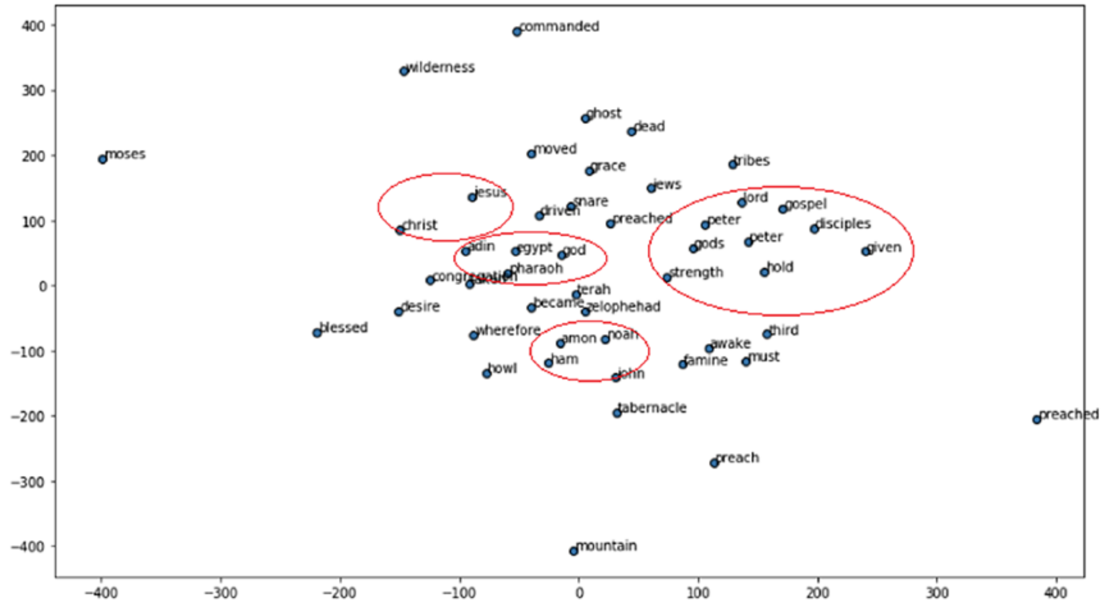


Figure 4-32. *Visualizing Skip-Gram Word2Vec word embeddings using t-SNE*

The circles in Figure 4-32 show different words of contextual similarity positioned near each other in the vector space. If you find any other interesting patterns, feel free to let me know!

Robust Word2Vec Models with Gensim

While our implementations are decent enough, they are not optimized to work well on large corpora. The *Gensim* framework, created by Radim Řehůřek, consists of a robust, efficient, and scalable implementation of the Word2Vec model. We will leverage this on our Bible corpus. In our workflow, we will tokenize our normalized corpus and then focus on the following four parameters in the Word2Vec model to build it. The basic idea is to provide a corpus of documents as input and get feature vectors for the output.

Internally, it constructs a vocabulary based on the input text documents and learns vector representations for words based on various techniques, which we mentioned earlier. Once this is complete, it builds a model that can be used to extract word vectors for each word in a document. Using various techniques like average weighting or TF-IDF weighting, we can compute the averaged vector representation of a document using its word vectors. You can get more details about the interface for Gensim's Word2Vec implementation at `http://radimrehurek.com/gensim/models/word2vec.html`. We will be mainly focusing on the following parameters when we build our model from our sample training corpus.

- `size`: This parameter is used to set the size or dimension for the word vectors and can range from tens to thousands. You can try various dimensions to see which gives the best result.
- `window`: This parameter is used to set the context or window size that specifies the length of the window of words that should be considered for the algorithm to take into account as context when training.
- `min_count`: This parameter specifies the minimum word count needed across the corpus for the word to be considered in the vocabulary. This helps remove very specific words that may not have much significance since they occur very rarely in the documents.
- `sample`: This parameter is used to downsample effects of occurrence of frequent words. Values between 0.01 and 0.0001 are usually ideal.

After building our model, we will use our words of interest to see the top similar words for each of them.

```
from gensim.models import word2vec

# tokenize sentences in corpus
wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_bible]

# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 30          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

w2v_model = word2vec.Word2Vec(tokenized_corpus, size=feature_size,
                          window=window_context, min_count=min_word_count,
                          sample=sample, iter=50)

# view similar words based on gensim's model
similar_words = {search_term: [item[0]
                     for item in w2v_model.wv.most_similar([search_term],
                     topn=5)]
                         for search_term in ['god', 'jesus', 'noah',
                         'egypt', 'john', 'gospel', 'moses','famine']}
similar_words

{'egypt': ['pharaoh', 'egyptians', 'bondage', 'rod', 'flowing'],
 'famine': ['pestilence', 'peril', 'blasting', 'mildew', 'morever'],
 'god': ['lord', 'promised', 'worldly', 'glory', 'reasonable'],
 'gospel': ['faith', 'afflictions', 'christ', 'persecutions', 'godly'],
 'jesus': ['peter', 'messias', 'apostles', 'immediately', 'neverthless'],
 'john': ['baptist', 'james', 'peter', 'galilee', 'zebedee'],
 'moses': ['congregation', 'children', 'aaron', 'ordinance', 'doctor'],
 'noah': ['shem', 'japheth', 'ham', 'noe', 'henoch']}
```

The similar words are more closely related to our words of interest and this is expected, given we ran this model for more iterations, which must yield better and more contextual embeddings. Do you notice any interesting associations? See Figure 4-33.

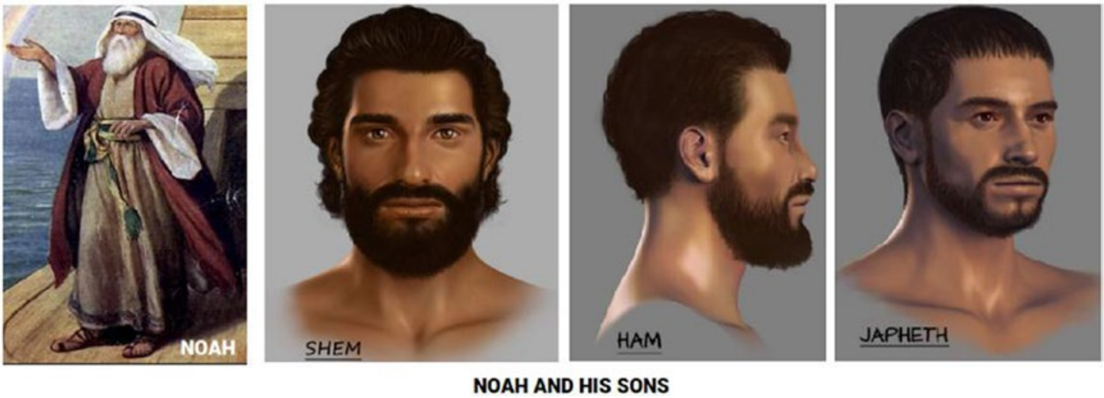


Figure 4-33. *Noah's sons come up as the most contextually similar entities from our model!*

Let's also visualize the words of interest and their similar words using their embedding vectors after reducing their dimensions to a 2D space with t-SNE. See Figure 4-34.

```
from sklearn.manifold import TSNE

words = sum([[k] + v for k, v in similar_words.items()], [])
wvs = w2v_model.wv[words]

tsne = TSNE(n_components=2, random_state=0, n_iter=10000, perplexity=2)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(wvs)
labels = words

plt.figure(figsize=(14, 8))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset
    points')
```

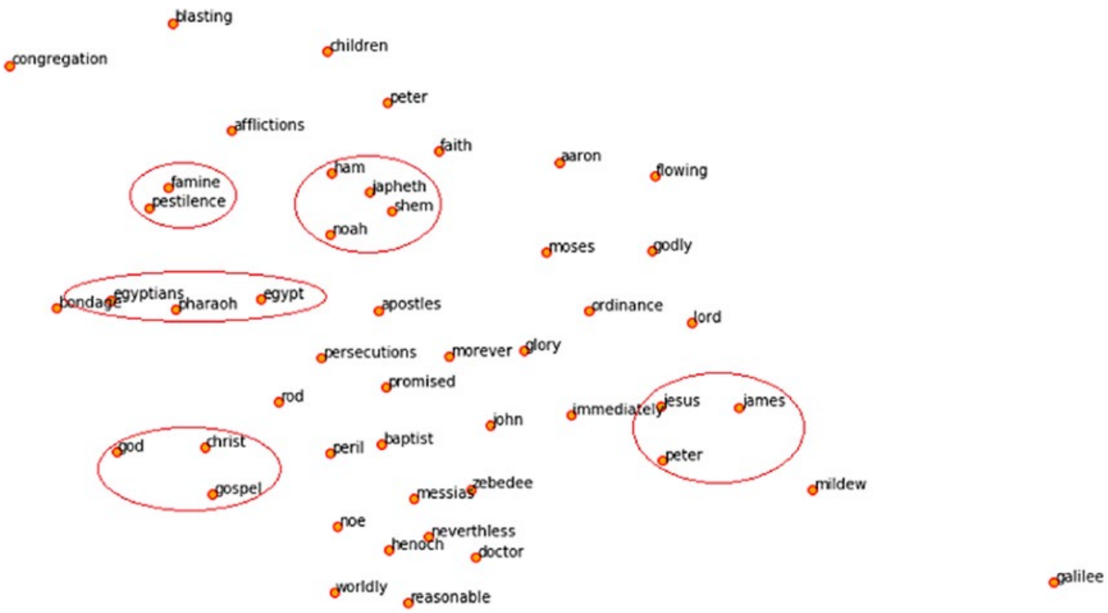


Figure 4-34. *Visualizing our Word2Vec word embeddings using t-SNE*

The circles have been drawn by me to point out some interesting associations. We can clearly see based on what I depicted earlier that Noah and his sons are quite close to each other based on the word embeddings from our model!

Applying Word2Vec Features for Machine Learning Tasks

If you remember from the previous section in this chapter, you might have seen me using features for some actual machine learning tasks like clustering. Let's leverage our other corpus and try to achieve this result. To start, we build a simple Word2Vec model on the corpus and visualize the embeddings. See Figure 4-35.

```
# build word2vec model
wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_corpus]

# Set values for various parameters
feature_size = 10    # Word vector dimensionality
window_context = 10          # Context window size
min_word_count = 1   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words
```

```
w2v_model = word2vec.Word2Vec(tokenized_corpus, size=feature_size,
                              window=window_context, min_count = min_word_
                              count, sample=sample, iter=100)

# visualize embeddings
from sklearn.manifold import TSNE
words = w2v_model.wv.index2word
wvs = w2v_model.wv[words]
tsne = TSNE(n_components=2, random_state=0, n_iter=5000, perplexity=2)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(wvs)
labels = words
plt.figure(figsize=(12, 6))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset
    points')
```

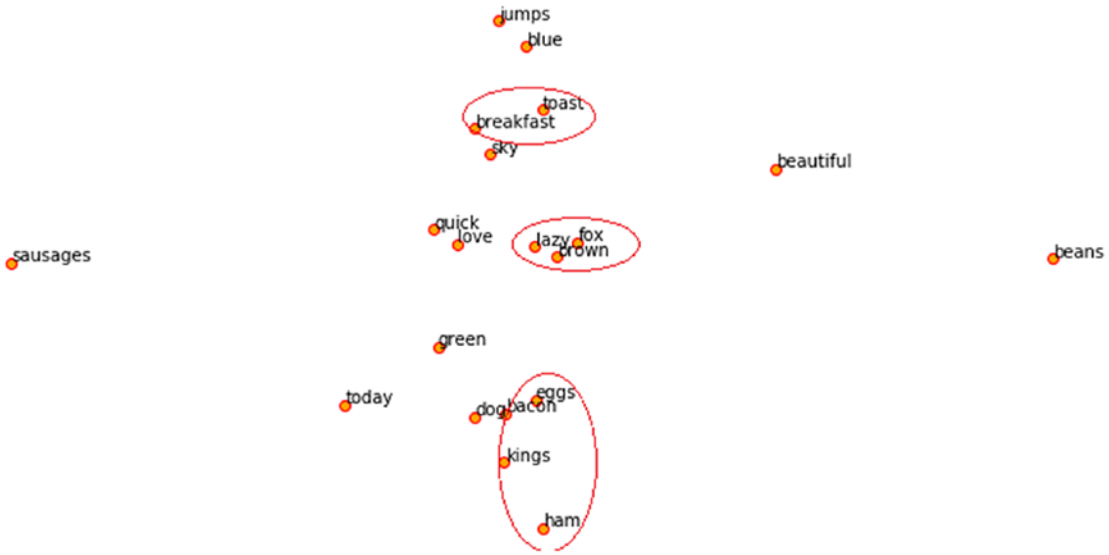


Figure 4-35. *Visualizing Word2Vec word embeddings on our other sample corpus*

Remember that our corpus is extremely small, so to get meaningful word embeddings and for the model to get more context and semantics, we need more data. Now what is a word embedding in this scenario? It's typically a dense vector for each word, as depicted in the following example for the word "sky".

```
w2v_model.wv['sky']
```

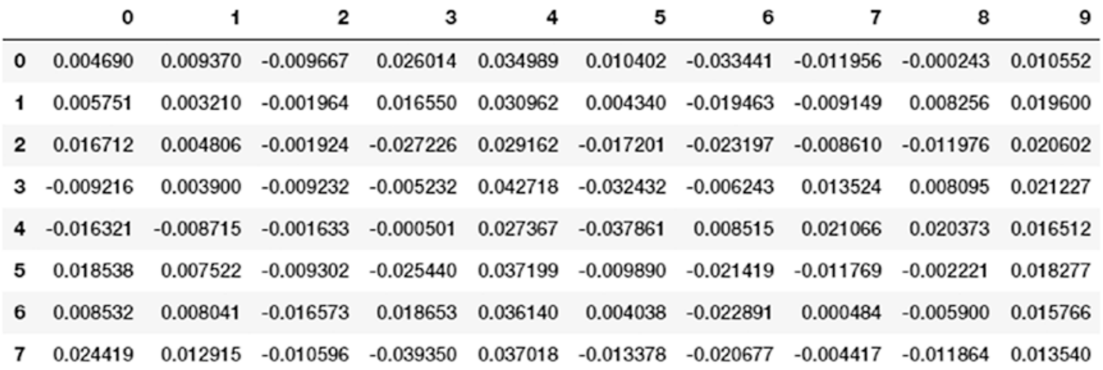
```
array([ 0.04576328,  0.02328374, -0.04483001,  0.0086611 ,  0.05173225,
        0.00953358, -0.04087641, -0.00427487, -0.0456274 ,  0.02155695],
        dtype=float32)
```

Strategy for Getting Document Embeddings

Now suppose we wanted to cluster the eight documents from our toy corpus. We would need to get the document-level embeddings from each of the words present in each document. One strategy would be to average the word embeddings for each word in a document. This is an extremely useful strategy and you can adopt it to your own problems. Let's apply this on our corpus to get features for each document. See Figure 4-36.

```
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector


def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary,
                  num_features) for tokenized_sentence in corpus]
    return np.array(features)
```

```
# get document level embeddings
w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus,
model=w2v_model, num_features=feature_size)
pd.DataFrame(w2v_feature_array)
```

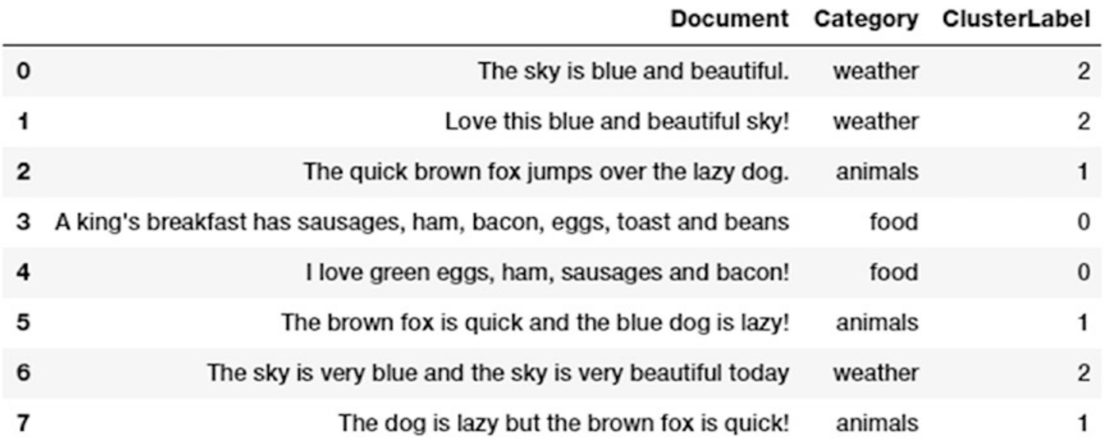


	0	1	2	3	4	5	6	7	8	9
0	0.004690	0.009370	-0.009667	0.026014	0.034989	0.010402	-0.033441	-0.011956	-0.000243	0.010552
1	0.005751	0.003210	-0.001964	0.016550	0.030962	0.004340	-0.019463	-0.009149	0.008256	0.019600
2	0.016712	0.004806	-0.001924	-0.027226	0.029162	-0.017201	-0.023197	-0.008610	-0.011976	0.020602
3	-0.009216	0.003900	-0.009232	-0.005232	0.042718	-0.032432	-0.006243	0.013524	0.008095	0.021227
4	-0.016321	-0.008715	-0.001633	-0.000501	0.027367	-0.037861	0.008515	0.021066	0.020373	0.016512
5	0.018538	0.007522	-0.009302	-0.025440	0.037199	-0.009890	-0.021419	-0.011769	-0.002221	0.018277
6	0.008532	0.008041	-0.016573	0.018653	0.036140	0.004038	-0.022891	0.000484	-0.005900	0.015766
7	0.024419	0.012915	-0.010596	-0.039350	0.037018	-0.013378	-0.020677	-0.004417	-0.011864	0.013540

Figure 4-36. *Document-level embeddings*

Now that we have our features for each document, let's cluster these documents using the affinity propagation algorithm, which is a clustering algorithm based on the concept of "message passing" between data points. It does not need the number of clusters as an explicit input, which is often required by partition-based clustering algorithms. This is discussed in more detail in Chapter 7. See Figure 4-37.

```
from sklearn.cluster import AffinityPropagation

ap = AffinityPropagation()
ap.fit(w2v_feature_array)
cluster_labels = ap.labels_
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```



	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	2
1	Love this blue and beautiful sky!	weather	2
2	The quick brown fox jumps over the lazy dog.	animals	1
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food	0
4	I love green eggs, ham, sausages and bacon!	food	0
5	The brown fox is quick and the blue dog is lazy!	animals	1
6	The sky is very blue and the sky is very beautiful today	weather	2
7	The dog is lazy but the brown fox is quick!	animals	1

***Figure 4-37.** Clusters assigned based on the document features from Word2Vec*

We can see that our algorithm has clustered each document into the right group based on our Word2Vec features. Pretty neat! We can also visualize how each document is positioned in each cluster by using *Principal Component Analysis (PCA)* to reduce the feature dimensions to 2D and then visualizing them (by color coding each cluster). See Figure 4-38.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2, random_state=0)
pcs = pca.fit_transform(w2v_feature_array)
labels = ap.labels_
categories = list(corpus_df['Category'])
plt.figure(figsize=(8, 6))

for i in range(len(labels)):
    label = labels[i]
    color = 'orange' if label == 0 else 'blue' if label == 1 else 'green'
    annotation_label = categories[i]
    x, y = pcs[i]
    plt.scatter(x, y, c=color, edgecolors='k')
    plt.annotate(annotation_label, xy=(x+1e-4, y+1e-3), xytext=(0, 0),
                 textcoords='offset points')
```

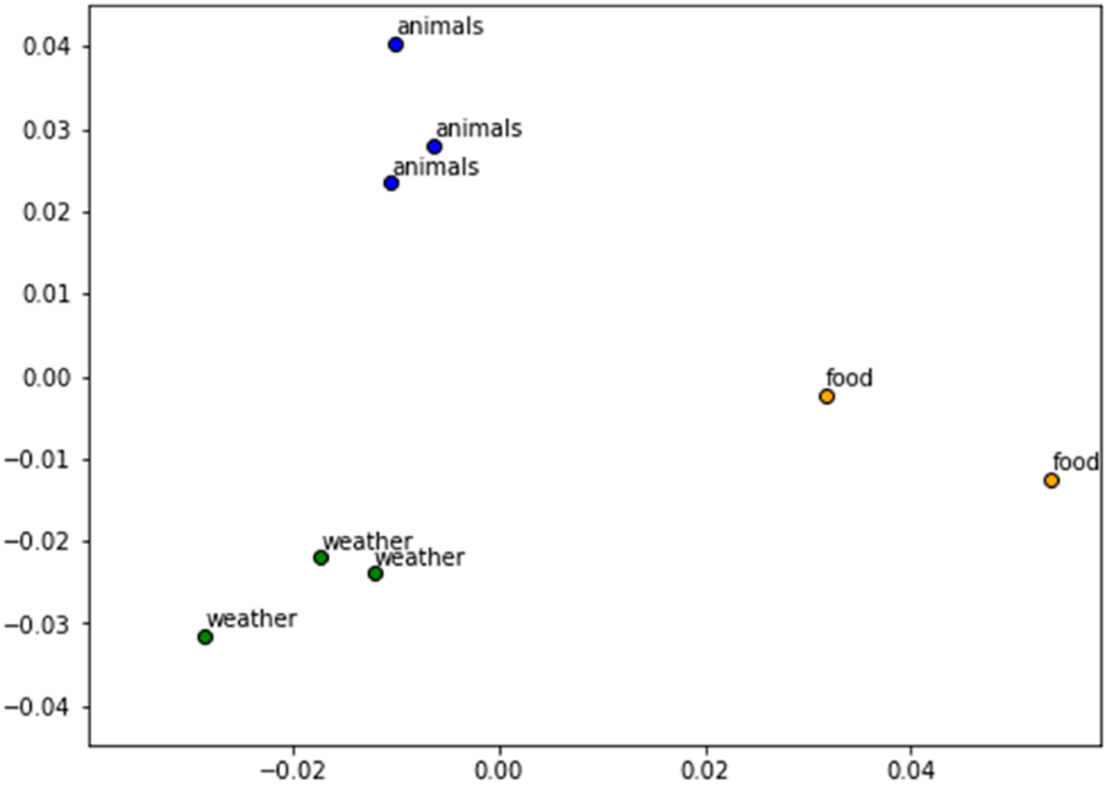


Figure 4-38. *Visualizing our document clusters*

Everything looks to be in order, as documents in each cluster are closer to each other and far apart from the other clusters.

The GloVe Model

The GloVe (*Global Vectors*) model is a unsupervised learning model that can be used to obtain dense word vectors similar to Word2Vec. However, the technique is different and training is performed on an aggregated global word-word co-occurrence matrix, giving us a vector space with meaningful sub-structures. This method was invented in Stanford by Pennington et al. and I recommend you read the original paper on GloVe, entitled “GloVe: Global Vectors for Word Representation,” by Pennington et al., which is an excellent read to get some perspective on how this model works.

We won't cover the implementation of the model from scratch in too much detail, but if you are interested in the actual code, you can check out the official GloVe page at `https://nlp.stanford.edu/projects/glove/`. We keep things simple here and try to understand the basic concepts behind the GloVe model.

We talked about count-based matrix factorization methods like LSA and predictive methods like Word2Vec. The paper claims that currently, both families suffer significant drawbacks. Methods like LSA efficiently leverage statistical information but they do relatively poorly on the word analogy task like how we found out semantically similar words. Methods like Skip-Gram may do better on the analogy task, but they poorly utilize the statistics of the corpus on a global level.

The basic methodology of the GloVe model is to first create a huge word-context co-occurrence matrix consisting of (word, context) pairs such that each element in this matrix represents how often a word occurs with the context (which can be a sequence of words). The idea then is to apply matrix factorization to approximate this matrix, as depicted in Figure 4-39.

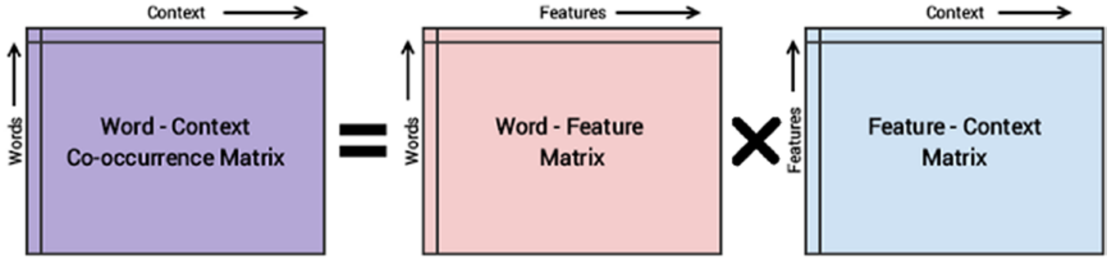


Figure 4-39. *Conceptual model for the GloVe model's implementation*

Considering the Word-Context (*WC*) matrix, Word-Feature (*WF*) matrix, and Feature-Context (*FC*) matrix, we try to factorize

$$\boldsymbol{WC} = \boldsymbol{WF} \times \boldsymbol{FC}$$

such that we we aim to reconstruct *WC* from *WF* and *FC* by multiplying them. For this, we typically initialize *WF* and *FC* with some random weights and attempt to multiply them to get *WC*′ (an approximation of WC) and measure how close it is to *WC*. We do this multiple times using *Stochastic Gradient Descent* (SGD) to minimize the error.

Finally, the *Word-Feature matrix* (*WF*) gives us the word embeddings for each word, where *F* can be preset to a specific number of dimensions. A very important point to remember is that both Word2Vec and GloVe models are very similar in how they work. Both of them aim to build a vector space where the position of each word is influenced by its neighboring words based on their context and semantics. Word2Vec starts with local individual examples of word co-occurrence pairs and GloVe starts with global aggregated co-occurrence statistics across all words in the corpus.

Applying *GloVe* Features for Machine Learning Tasks

Let's try to leverage GloVe-based embeddings for our document clustering task. The very popular *spaCy* framework comes with capabilities to leverage GloVe embeddings based on different language models. You can also get pretrained word vectors from Stanford NLP's website (https://nlp.stanford.edu/projects/glove/) and load them as needed using Gensim or spaCy. We will install spaCy and use the en_vectors_web_lg model (https://spacy.io/models/en#en_vectors_web_lg), which consists of 300-dimensional word vector dense embeddings trained on the Common Crawl (http://commoncrawl.org/) with GloVe.

```
# Use the following command to install spaCy
> pip install -U spacy
OR
> conda install -c conda-forge spacy
# Download the following language model and store it in disk
https://github.com/explosion/spacy-models/releases/tag/en_vectors_web_lg-2.0.0
# Link the same to spacy
> python -m spacy link ./spacymodels/en_vectors_web_lg-2.0.0/en_vectors_
web_lg en_vecs
Linking successful
    ./spacymodels/en_vectors_web_lg-2.0.0/en_vectors_web_lg -->
    ./Anaconda3/lib/site-packages/spacy/data/en_vecs
You can now load the model via spacy.load('en_vecs')
```

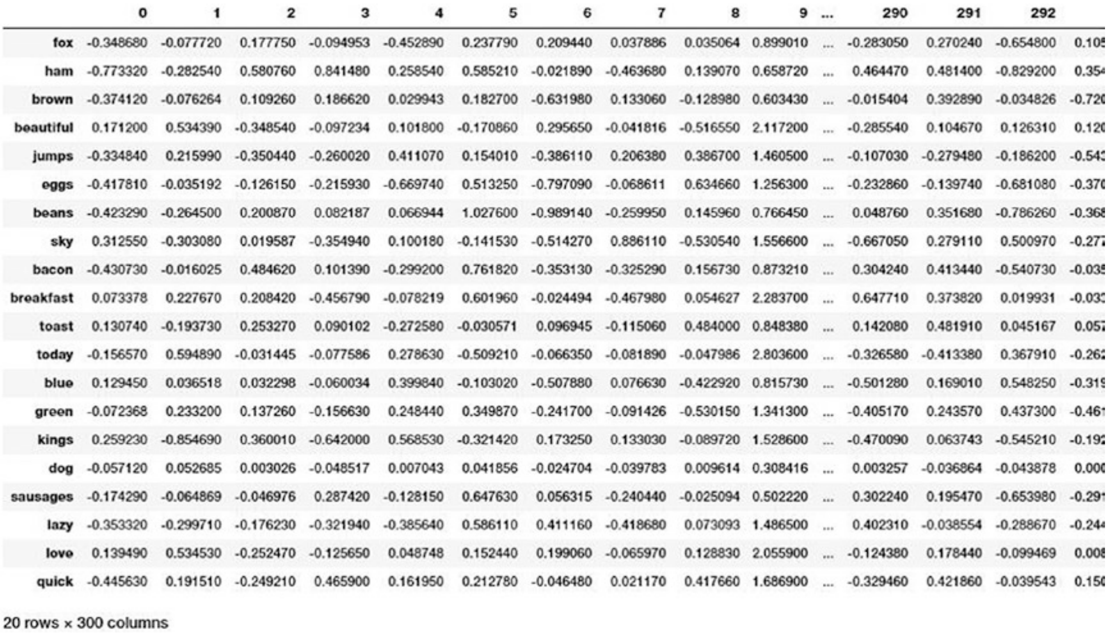
There are automated ways to install models in spaCy too. You can check the Models & Languages page at https://spacy.io/usage/models for more information if needed. I had some issues with it, so I had to manually load them. We now load our language model using spaCy.

```
import spacy

nlp = spacy.load('en_vecs')
total_vectors = len(nlp.vocab.vectors)
print('Total word vectors:', total_vectors)
```

```
Total word vectors: 1070971
```

This validates that everything is working and in order. Let's get the GloVe embeddings for each of our words now in our toy corpus. See Figure 4-40.

```
unique_words = list(set([word for sublist in [doc.split() for doc in norm_
corpus] for word in sublist]))
word_glove_vectors = np.array([nlp(word).vector for word in unique_words])
pd.DataFrame(word_glove_vectors, index=unique_words)
```



	0	1	2	3	4	5	6	7	8	9	...	290	291	292	
fox	-0.348680	-0.077720	0.177750	-0.094953	-0.452890	0.237790	0.209440	0.037886	0.035064	0.899010	...	-0.283050	0.270240	-0.654800	0.105
ham	-0.773320	-0.282540	0.580760	0.841480	0.258540	0.585210	-0.021890	-0.463680	0.139070	0.658720	...	0.464470	0.481400	-0.829200	0.354
brown	-0.374120	-0.076264	0.109260	0.186620	0.029943	0.182700	-0.631980	0.133060	-0.128980	0.603430	...	-0.015404	0.392890	-0.034826	-0.720
beautiful	0.171200	0.534390	-0.348540	-0.097234	0.101800	-0.170860	0.295650	-0.041816	-0.516550	2.117200	...	-0.285540	0.104670	0.126310	0.120
jumps	-0.334840	0.215990	-0.350440	-0.260020	0.411070	0.154010	-0.386110	0.206380	0.386700	1.460500	...	-0.107030	-0.279480	-0.186200	-0.543
eggs	-0.417810	-0.035192	-0.126150	-0.215930	-0.669740	0.513250	-0.797090	-0.068611	0.634660	1.256300	...	-0.232860	-0.139740	-0.681080	-0.370
beans	-0.423290	-0.264500	0.200870	0.082187	0.066944	1.027600	-0.989140	-0.259950	0.145960	0.766450	...	0.048760	0.351680	-0.786260	-0.368
sky	0.312550	-0.303080	0.019587	-0.354940	0.100180	-0.141530	-0.514270	0.886110	-0.530540	1.556600	...	-0.667050	0.279110	0.500970	-0.277
bacon	-0.430730	-0.016025	0.484620	0.101390	-0.299200	0.761820	-0.353130	-0.325290	0.156730	0.873210	...	0.304240	0.413440	-0.540730	-0.035
breakfast	0.073378	0.227670	0.208420	-0.456790	-0.078219	0.601960	-0.024494	-0.467980	0.054627	2.283700	...	0.647710	0.373820	0.019931	-0.033
toast	0.130740	-0.193730	0.253270	0.090102	-0.272580	-0.030571	0.096945	-0.115060	0.484000	0.848380	...	0.142080	0.481910	0.045167	0.057
today	-0.156570	0.594890	-0.031445	-0.077586	0.278630	-0.509210	-0.066350	-0.081890	-0.047986	2.803600	...	-0.326580	-0.413380	0.367910	-0.262
blue	0.129450	0.036518	0.032298	-0.060034	0.399840	-0.103020	-0.507880	0.076630	-0.422920	0.815730	...	-0.501280	0.169010	0.548250	-0.319
green	-0.072368	0.233200	0.137260	-0.156630	0.248440	0.349870	-0.241700	-0.091426	-0.530150	1.341300	...	-0.405170	0.243570	0.437300	-0.461
kings	0.259230	-0.854690	0.360010	-0.642000	0.568530	-0.321420	0.173250	0.133030	-0.089720	1.528600	...	-0.470090	0.063743	-0.545210	-0.192
dog	-0.057120	0.052685	0.003026	-0.048517	0.007043	0.041856	-0.024704	-0.039783	0.009614	0.308416	...	0.003257	-0.036864	-0.043878	0.000
sausages	-0.174290	-0.064869	-0.046976	0.287420	-0.128150	0.647630	0.056315	-0.240440	-0.025094	0.502220	...	0.302240	0.195470	-0.653980	-0.291
lazy	-0.353320	-0.299710	-0.176230	-0.321940	-0.385640	0.586110	0.411160	-0.418680	0.073093	1.486500	...	0.402310	-0.038554	-0.288670	-0.244
love	0.139490	0.534530	-0.252470	-0.125650	0.048748	0.152440	0.199060	-0.065970	0.128830	2.055900	...	-0.124380	0.178440	-0.099469	0.008
quick	-0.445630	0.191510	-0.249210	0.465900	0.161950	0.212780	-0.046480	0.021170	0.417660	1.686900	...	-0.329460	0.421860	-0.039543	0.150

20 rows × 300 columns

Figure 4-40. *GloVe embeddings for words in our sample corpus*

We can now use t-SNE to visualize these embeddings, similar to what we did using our Word2Vec embeddings. See Figure 4-41.

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=0, n_iter=5000, perplexity=3)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(word_glove_vectors)
labels = unique_words

plt.figure(figsize=(12, 6))
plt.scatter(T[:, 0], T[:, 1], c='orange', edgecolors='r')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset
    points')
```

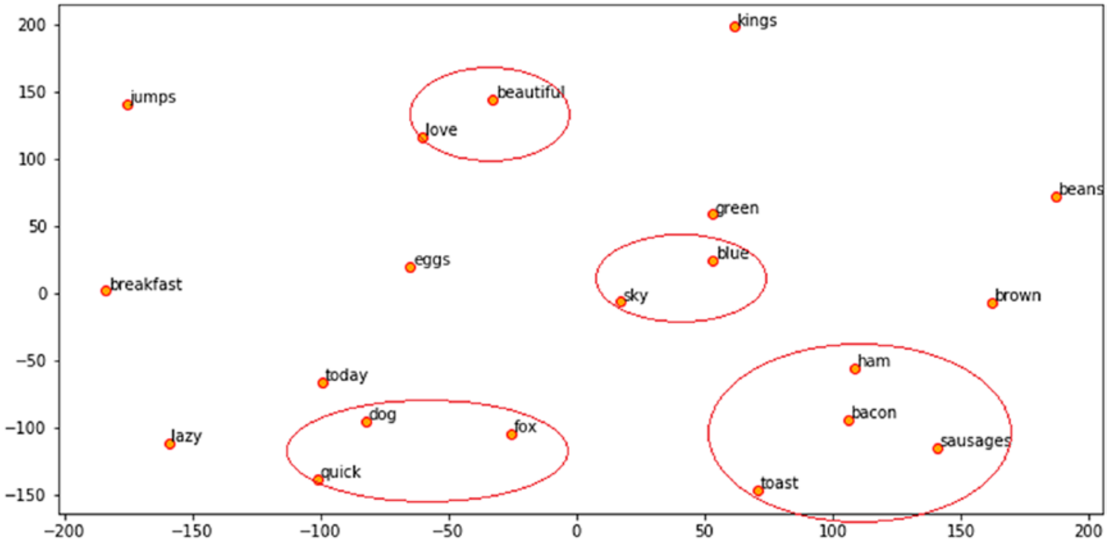
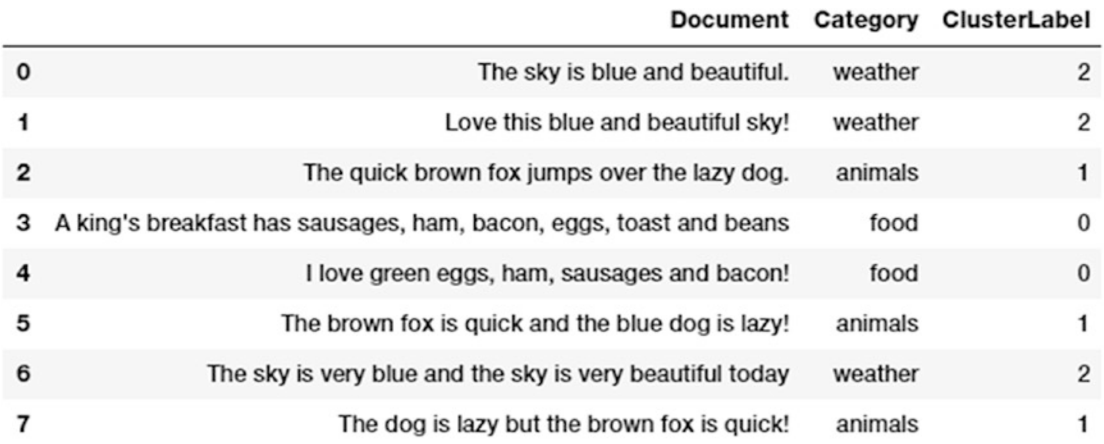


Figure 4-41. *Visualizing GloVe word embeddings on our sample corpus*

The beauty of spaCy is that it automatically provides the averaged embeddings for words in each document without us having to implement a function like we did in Word2Vec. We will now leverage spaCy to get document features for our corpus and use k-means clustering to cluster our documents. See Figure 4-42.

```
doc_glove_vectors = np.array([nlp(str(doc)).vector for doc in norm_corpus])

km = KMeans(n_clusters=3, random_state=0)
km.fit_transform(doc_glove_vectors)
cluster_labels = km.labels_
cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
pd.concat([corpus_df, cluster_labels], axis=1)
```



	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	2
1	Love this blue and beautiful sky!	weather	2
2	The quick brown fox jumps over the lazy dog.	animals	1
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food	0
4	I love green eggs, ham, sausages and bacon!	food	0
5	The brown fox is quick and the blue dog is lazy!	animals	1
6	The sky is very blue and the sky is very beautiful today	weather	2
7	The dog is lazy but the brown fox is quick!	animals	1

Figure 4-42. *Clusters assigned based on our document features from GloVe*

We see consistent clusters similar to what we obtained from our Word2Vec model, which is good! The GloVe model claims to perform better than the Word2Vec model in many scenarios, as illustrated in Figure 4-43, which is from the original paper by Pennington et al.

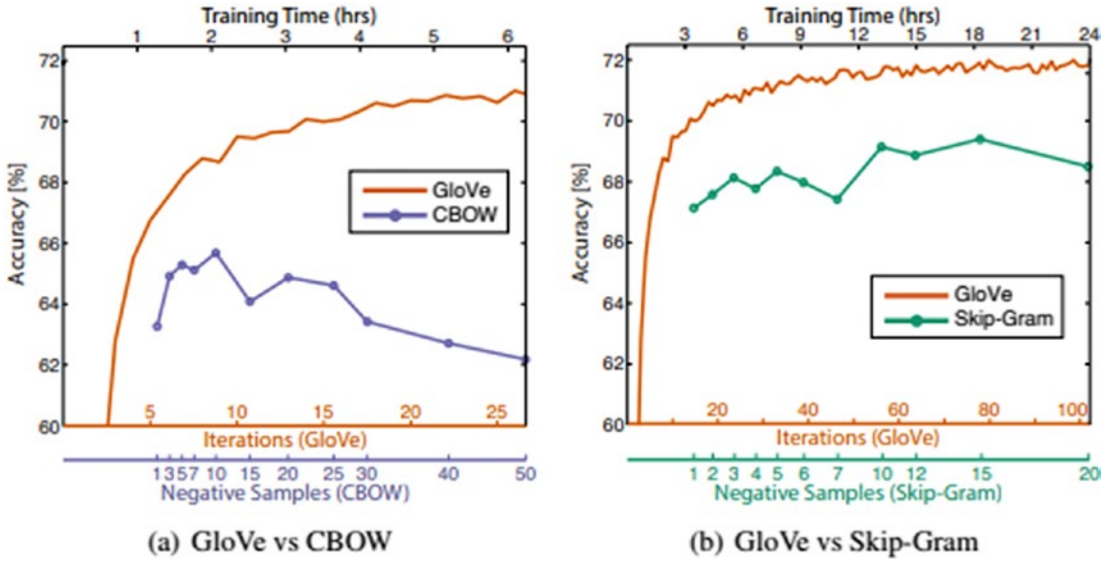


Figure 4-43. *GloVe vs Word2Vec performance (Source: https://nlp.stanford.edu/pubs/glove.pdf by Pennington et al.)*

These experiments were done by training 300-dimensional vectors on the same 6B token corpus (Wikipedia 2014 + Gigaword 5) with the same 400,000 word vocabulary and a symmetric context window of size 10 (in case you are interested in the details).

The FastText Model

The *FastText* model was introduced by Facebook in 2016 as an extension and supposedly improvement of the vanilla Word2Vec model. It's based on the original paper entitled "Enriching Word Vectors with Subword Information" by Mikolov et al., which is an excellent read to gain in-depth understanding into how this model works. Overall, FastText is a framework for learning word representations and performing robust, fast, and accurate text classifications. The framework is open sourced by Facebook on GitHub and claims to have the following.

- Recent state-of-the-art English word vectors
- Word vectors for 157 languages trained on Wikipedia and Crawl
- Models for language identification and various supervised tasks

Although I haven't implemented this model from scratch, based on the research paper, the following is what I learned about how the model works. In general, predictive models like Word2Vec consider each word a distinct entity (e.g., where) and generate a dense embedding for the word. However, this is a serious limitation with languages that have massive vocabularies and many rare words. The Word2Vec model typically ignores the morphological structure of each word and considers a word a single entity. The FastText model considers each word a Bag of Character n-grams. This is also called a *subword* model in the paper.

We add special boundary symbols < and > at the beginning and end of words. This enables us to distinguish prefixes and suffixes from other character sequences. We also include the letter `w` in the set of its n-grams to learn a representation for each word (in addition to its character n-grams). Taking the word "where" and *n=3* (tri-grams) as an example, it will be represented by the character n-grams: <wh, whe, her, ere, re> and the special sequence <where>, which represents the whole word. Note that the sequence corresponding to the word <her> is different from the tri-gram "her" and the word "where".

In practice, the paper recommends extracting all the n-grams for n ≥ 3 and n ≤ 6. This is a very simple approach, and different sets of n-grams could be considered, for example taking all prefixes and suffixes. We typically associate a vector representation (embedding) to each n-gram for a word. Thus, we can represent a word by the sum of the vector representations of its n-grams or the average of the embedding of these n-grams. Thus, due to this effect of leveraging n-grams from individual words based on their characters, there is a higher chance for rare words to get good representation since their character-based n-grams should occur across other words of the corpus.

Applying FastText Features to Machine Learning Tasks

The Gensim package has wrappers that provide interfaces to leverage the FastText model available under the `gensim.models.fasttext` module. Let's apply this once again to the Bible corpus and look at the words of interest and their most similar words.

```
from gensim.models.fasttext import FastText

wpt = nltk.WordPunctTokenizer()
tokenized_corpus = [wpt.tokenize(document) for document in norm_bible]
```

```
# Set values for various parameters
feature_size = 100    # Word vector dimensionality
window_context = 50          # Context window size
min_word_count = 5   # Minimum word count
sample = 1e-3   # Downsample setting for frequent words

# sg decides whether to use the skip-gram model (1) or CBOW (0)
ft_model = FastText(tokenized_corpus, size=feature_size, window=window_
context, min_count=min_word_count,sample=sample, sg=1, iter=50)

# view similar words based on gensim's FastText model
similar_words = {search_term: [item[0]
                    for item in ft_model.wv.most_similar([search_term],
                    topn=5)]
                        for search_term in ['god', 'jesus', 'noah',
                        'egypt', 'john', 'gospel', 'moses','famine']}
similar_words

{'egypt': ['land', 'pharaoh', 'egyptians', 'pathros', 'assyrian'],
 'famine': ['pestilence', 'sword', 'egypt', 'dearth', 'blasted'],
 'god': ['lord', 'therefore', 'jesus', 'christ', 'truth'],
 'gospel': ['preached', 'preach', 'christ', 'preaching', 'gentiles'],
 'jesus': ['christ', 'god', 'disciples', 'paul', 'grace'],
 'john': ['baptist', 'baptize', 'peter', 'philip', 'baptized'],
 'moses': ['aaron', 'commanded', 'congregation', 'spake', 'tabernacle'],
 'noah': ['shem', 'methuselah', 'creepeth', 'adam', 'milcah']}
```

You can see a lot of similarity in the results (see Figure 4-44). Do you notice any interesting associations and similarities?

***Figure 4-44.** Moses, his brother Aaron, and the Tabernacle of Moses come up as similar entities from our model*

Having these embeddings, we can perform some interesting natural language tasks. One of these is to determine the similarity between different words (entities).

```
print(ft_model.wv.similarity(w1='god', w2='satan'))
print(ft_model.wv.similarity(w1='god', w2='jesus'))
```

```
0.333260876685
0.698824900473
```

```
st1 = "god jesus satan john"
print('Odd one out for [',st1, ']:', ft_model.wv.doesnt_match(st1.split()))
st2 = "john peter james judas"
print('Odd one out for [',st2, ']:', ft_model.wv.doesnt_match(st2.split()))
```

```
Odd one out for [ god jesus satan john ]: satan
Odd one out for [ john peter james judas ]: judas
```

We can see that “god” is more closely associated with “jesus” than “satan,” based on the text in the Bible corpus. Similar results can be seen in both cases for the odd entity among the other words.

Summary

We covered a wide variety of feature engineering techniques and models in this chapter. We covered traditional and advanced, newer models of text representation. Remember that traditional strategies are based on concepts from mathematics, information retrieval, and natural language processing. Hence, these tried and tested methods over time have proven to be successful in a wide variety of datasets and problems. We covered a wide variety of traditional feature engineering models, including the Bag of Words, Bag of N-Grams, TF-IDF, similarity, and topic models. We also implemented some models from scratch to better understand the concepts with hands-on examples.

Traditional models have some limitations considering sparse representations, leading to feature explosion. This causes the curse of dimensionality and losing context, ordering, and sequence of related words in text data. This is where we covered advanced feature engineering models, which leverage deep learning and neural network models to generate dense embeddings for every word in any corpus.

We took a deep dive into Word2Vec and even trained deep learning models from scratch to showcase how the CBOW and Skip-Gram models work. Understanding how to use a feature engineering model in the real world is also important and we demonstrated how to extract and build document-level features and use them for text clustering. Finally, we covered essential concepts and detailed examples of two other advanced feature engineering models—GloVe and FastText. We encourage you to try leveraging these models in your own problems. We also use these models in the next chapter on text classification!

CHAPTER 5

Text Classification

Learning to process and understand text is one of the first, yet most essential, steps on the journey to getting meaningful insights from textual data. While it is important to understand language syntax, structure, and semantics, it is not sufficient on its own to be able to derive useful patterns and insights and get maximum use out of vast volumes of text data. Knowledge of language processing coupled with concepts from artificial intelligence, machine learning, and deep learning help in building intelligent systems, which can leverage text data and help solve real-world practical problems that benefit businesses and enterprises.

There are various aspects in machine learning, which include supervised learning, unsupervised learning, reinforcement learning, and more recently, deep learning. Each of these domains have several techniques and algorithms, which can be leveraged on top of text data and thus enable us to build self-learning systems, which do not need too much manual supervision. A machine learning model is a combination of data and algorithms and we got a taste of them in Chapter 3 when we were building our own parsers and taggers. The benefit of machine learning is that once a model is trained, we can directly use that model on new and previously unseen data to start seeing useful insights and desired results—the key to predictive and prescriptive analytics!

One of the most relevant and challenging problems in the domain of natural language processing is *text classification* or categorization, also popularly known as *document classification*. This task involves categorizing or classifying text documents into various (predefined) categories based on inherent properties or attributes of each text document. This has applications in diverse domains and businesses, including email spam identification and news categorization. The concept might seem simple and if you have a small number of documents, you can look at each document and gain some idea about what it is trying to indicate. Based on this knowledge, you can group similar documents into categories or classes. It starts getting more challenging once the number of text documents to be classified increases to several hundred thousands or millions. This is where techniques like feature extraction and supervised or unsupervised

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_5

machine learning come in handy. Document classification is a generic problem not limited to text alone but also can be extended for other items like music, images, video, and other media.

To formalize this problem more clearly, we will have a given set of classes and several text documents. Remember that documents are basically sentences or paragraphs of text. This forms a corpus. Our task is to determine which class or classes each document belongs to. This entire process involves several steps, which we will be discussing in more detail shortly. Briefly, for a supervised classification problem, we need to have some labeled data that we can use for training a text classification model. This data is essentially curated documents that are already assigned to some specific class or category beforehand. Using this, we essentially extract features and attributes from each document and make our model learn these attributes corresponding to each particular document and its class/category. This is done by feeding it to a supervised machine learning algorithm.

Of course the data needs to be preprocessed and normalized before building the model. Once done, we follow the same process of normalization and feature extraction and then feed it to the model to predict the class or category for new documents. However, for an unsupervised classification problem, we essentially do not have any labeled training documents and instead use techniques like clustering and document similarity measures to cluster documents based on their inherent properties and assign labels to them.

In this chapter, we discuss the concept of text document classification and learn how it can be formulated as a supervised machine learning problem. We also talk about the various forms of classification and what they indicate. A clear depiction of the essential steps necessary to complete a text classification workflow are also presented and we cover the essential steps from the same workflow. Some of these we covered in Chapters 3 and 4, including text wrangling and feature engineering and newer aspects including supervised machine learning classifiers, model evaluation, and tuning. Finally we put all these components together to build an end-to-end text classification system. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at `https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition`.

What Is Text Classification?

Before we define text classification, we need to understand the scope of textual data and what we mean by classification. The textual data involved here can be anything ranging from a phrase, a sentence, or a complete document with paragraphs of text that can be obtained from corpora, blogs, anywhere from the web, or even an enterprise data warehouse. Text classification is also often called *document classification* just to cover all forms of textual content under the term "document". While the term "document" can be defined as some form of concrete representation of thoughts or events, which could be in the form of writing, recorded speech, drawings, or presentations, we use the term "document" to represent textual data like sentences or paragraphs belonging to the English language (feel free to extend this to other languages as long as you are able to parse and process that language!).

Text classification is also often called *text categorization*. However, we explicitly use the word "classification" for two reasons. The first reason is because it depicts the same essence as text categorization, where we want to classify documents. The second reason is to depict that we are using *classification* or a *supervised machine learning approach* to classify or categorize text. Text categorization can be done in many ways. We focus explicitly on using a supervised approach using classification. The process of classification is not restricted to text alone and is used quite frequently in other domains like science, healthcare, weather, and technology.

Formal Definition

Now that we have all our background assumptions cleared, we can formally define the task of text classification and its overall scope. Text or document classification is defined as the process of assigning text documents into one or more classes or categories, assuming that we have a predefined set of classes. Documents are textual documents and each document can contain a sentence or even a paragraph of words. A text classification system can successfully classify each document to its correct class(es) based on inherent properties of the document.

Mathematically, we can define it as, given some description and attributes **d** for a document **D**, where $d \in D$, and given a set of predefined classes or categories, $C = \{c_1, c_2, c_3, \ \dots, c_n\}$. The actual document **D** can have many inherent properties and attributes, which lead it to being an entity in a high-dimensional space. Using a subset of that space with a limited set of descriptions and features depicted by **d**, we should be able to successfully assign the original document, **D** to its correct class C_x using a text classification system **T**. This can be represented by $T : D \rightarrow C_x$. We talk more about the text classification system in detail in subsequent sections. Figure 5-1 shows a high-level conceptual representation of the text classification process.

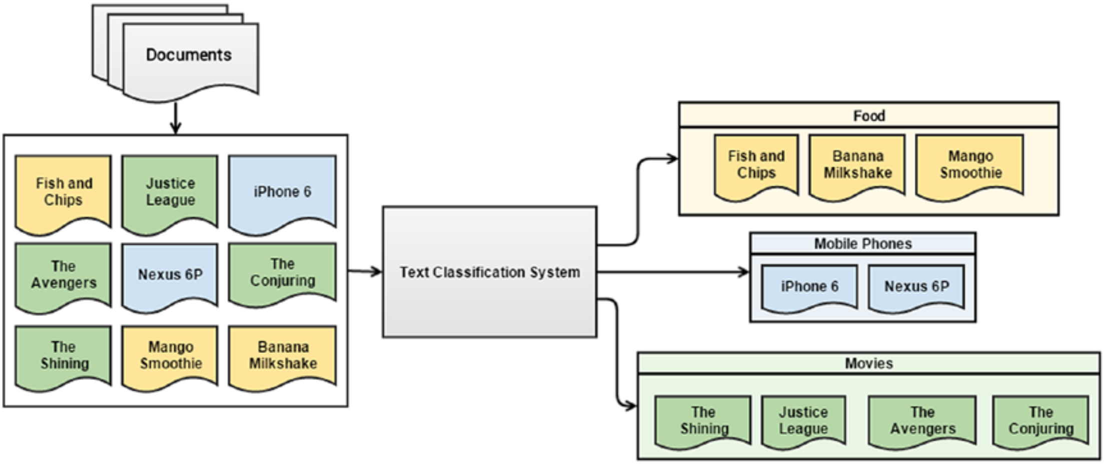


Figure 5-1. *Conceptual overview of text classification*

In Figure 5-1, we can see there are several documents that can be assigned to various categories of food, mobile phones, and movies. Initially, these documents are all present together, just like a text corpus has various documents in it. Once it goes through a text classification system, which is represented as a black box, we can see that each document is assigned to one specific class or category we defined previously. The documents are just represented by their names in real data; they can contain descriptions of each product, specific attributes like movie genre, product specifications, constituents, and much more, which are basically properties that can be used as features in the text classification system, to make document identification and classification easier.

Major Text Classification Variants

There are various types of text classification. We mention two major types that are based on the type of content that make up the documents. They are as follows.

- Content-based classification
- Request-based classification

These are more like different philosophies behind approaches to classifying text documents rather than specific technical algorithms or processes. *Content-based classification* is the type of text classification where priorities or weights are given to specific subjects or topics in the text content, which help determine the class of the document. A conceptual example is that of a book with more than 30% of the content about food preparation. It can be classified under cooking/recipes. *Request-based classification* is influenced based on user requests and is targeted toward specific user groups and audience. This type of classification is governed by specific policies and ideals based on user behavior and decisions.

Automated Text Classification

We have an idea of the definition and scope of text classification. We have also formally defined text classification conceptually and mathematically, where we talked about a "text classification system" being able to classify text documents to their respective categories or classes. Consider several humans going through each document and classifying it. They would then be a part of the text classification system that we are talking about. However, that would not scale very well once we had millions of text documents to be classified in short time intervals. To make the process more efficient and faster, we can automate the task of text classification, which brings us to automated text classification. To automate text classification, we use several machine learning techniques and concepts. There are two main types of machine learning techniques that are relevant to solving this problem. They are as follows:

- Supervised machine learning
- Unsupervised machine learning

Besides these two techniques, there are also other families of learning algorithms like *reinforcement learning* and *semi-supervised learning*. We'll look at supervised and unsupervised learning algorithms in more detail from both a machine learning perspective as well as how they can be leveraged in classifying text documents.

Unsupervised learning refers to specific machine learning techniques or algorithms that do not require any prelabeled training data samples to build a model. The focus is more on pattern mining and finding latent substructures in the data rather than predictive analytic'. We usually have a collection of data points, which could be textual or numeric depending on the problem we are trying to solve. We extract features from each of the data points using a process known as *feature engineering* and then we feed the feature set for each data point into our algorithm and try to extract meaningful patterns from the data, like trying to group together similar data points using techniques like clustering or summarizing documents based on topic models.

This is extremely useful in text document categorization and is also called *document clustering*, where we cluster documents into groups based on their features, similarity, and attributes, without training any model on previously labeled data. We discuss unsupervised learning more in future chapters when we cover topic models, document summarization, similarity analysis, and clustering.

Supervised learning refers to specific machine learning techniques or algorithms that are trained on prelabeled data samples, known as training data, and corresponding training labels/classes. Features are extracted from this data using feature engineering and each data point has its own feature set and corresponding class/label. The algorithm learns various patterns for each type of class from the training data. Once this process is complete, we have a trained model. This model can then be used to predict the class for future test data samples once we feed their features to the model. Thus, the machine has actually learned, based on previous training data samples, how to predict the class for new unseen data samples. There are two major types of supervised learning algorithms, described as follows:

- **Classification**: Supervised learning algorithms are known as classification when the outcomes to be predicted are distinct categories, thus the outcome variable is a categorical variable in this case. Examples are news categories and movie genres.
- **Regression**: Supervised learning algorithms are known as regression algorithms when the outcome we want to predict is a continuous numeric variable. Examples are house prices and weather temperature.

We specifically focus on classification for our problem because the name of the chapter speaks for itself. We are trying to classify text documents into distinct classes. We follow supervised learning approaches using different classification models in our implementations.

Formal Definition

Now we are ready to mathematically define the process of automated or machine learning based text classification. Consider we now have a training set of documents labeled with their corresponding class or category. This can be represented by **TS**, which is a set of paired documents and labels, $TS = \{(d_1, c_1), (d_2, c_2), \ldots, (d_n, c_n)\}$, where $d_1, d_2, \ldots, d_n$ is the list of text documents. Their corresponding labels are $c_1, c_2, \ldots, c_n$ such that $c_x \in C = \{c1, c2, \ldots, cn\}$ where c_x denoted the class label for document **x** and **C** denotes the set of all possible distinct classes, any of which can be the class for each document. Assuming we have our training set, we can define a supervised learning algorithm **F** such that, when it is trained on our training dataset **TS**, we build a classification model or classifier **γ** and $F(TS) = \gamma$. Thus, the supervised learning algorithm **F** takes the input set of (document, class) pairs **TS** and gives us the trained classifier **γ**, which is our model. This process is known as the training process. This model can then take a new, previously unseen document **ND** and predict its class c_{ND} such that $c_{ND} \in C$. This process is known as the *prediction process* and can be represented by $\gamma : TD \rightarrow c_{ND}$. Thus, we can see that there are two main stages in the supervised text classification process:

- Training
- Prediction

An important point to remember is that some manually labeled training data is necessary for supervised text classification so even though we are talking about automated text classification, to kick start the process, we need some manual effort. Of course the benefits of this are manifold since, once we have a trained classifier, we can keep using it to predict and classify new documents with minimal effort and manual supervision. There are various learning methods or algorithms, which we discuss in a future section. These learning algorithms are not specific just for text data but are generic machine learning algorithms that can be applied to various types of data after due preprocessing and feature engineering. We touch upon several supervised machine learning algorithms and use them in solving our real-world text classification problem.

These algorithms are usually trained on the training dataset and often an optional validation set so that the trained model does not overfit to the training data, which basically means it would then not be able to generalize well and predict properly for new instances of text documents. The model is often tuned based on several of its internal parameters (known as *hyperparameters*) based on the learning algorithm and by evaluating various performance metrics like accuracy on the validation set or by using cross-validation, where we split the training dataset into training and validation sets by random sampling. This is comprised of the training process whose outcome yields a fully trained model that's ready to predict. In the prediction stage, we usually have new data points from the test dataset. We can use them to feed into the model after normalization and feature engineering and see how well the model is performing by evaluating its prediction performance.

Text Classification Task Variants

There are several variants of text classification tasks, based on the number of classes to predict and the nature of predictions. They are as follows:

- Binary classification
- Multi-class classification
- Multi-label classification

These types of classification are based on the dataset, the number of classes/categories pertaining to that dataset, and the number of classes that can be predicted on any data point. *Binary classification* is when the total number of distinct classes or categories is two and any prediction can contain either one of those classes. *Multi-class classification* is also known as multinomial classification and refers to a problem where the total number of classes is more than two and each prediction gives one class or category, which can belong to any of those classes. This this is an extension of the binary classification problem, where the total number of classes is more than two. *Multi-label classification* refers to problems where each prediction can yield more than one outcome/predicted class for any data point.

Text Classification Blueprint

Now that we know the basic scope that the automated text classification entails, we present a blueprint for a complete workflow of building an automated text classifier system in this section. This consists of a series of steps that must be followed in the training and testing phases. To build a text classification system, we need to make sure we have our source of data and retrieve that data so that we can start feeding it to our system. The following main steps outline a typical workflow for a text classification system, assuming that we have our dataset already downloaded and ready to be used.

- Prepare train and test datasets (optionally a validation dataset)
- Preprocess and normalize text documents
- Feature extraction and engineering
- Model training
- Model prediction and evaluation
- Model deployment

These are the main steps that are carried out in that order for building a text classifier. Figure 5-2 shows a detailed workflow for a text classification system with the main components highlighted in training and prediction.

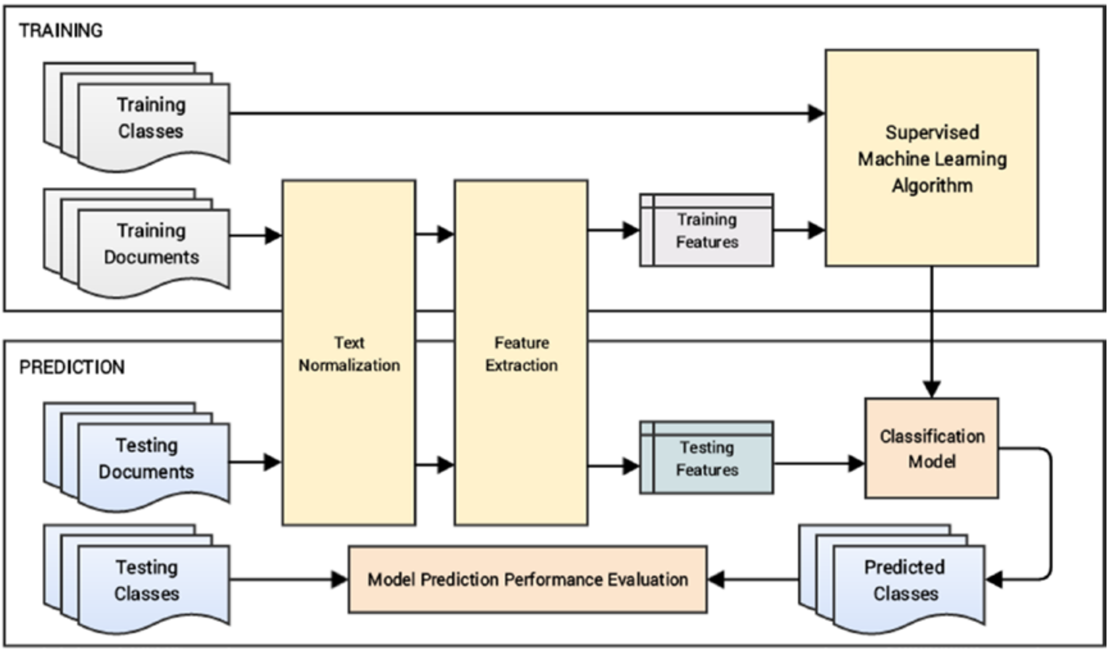


Figure 5-2. *Blueprint for building an automated text classification system*

From Figure 5-2, we notice that there are two main boxes called training and prediction, which are the two main stages involved in building a text classifier. In general, the dataset that we have is usually divided into two or three splits called the training, validation (optional), and testing datasets. Notice an overlap of the "Text Normalization" module and the "Feature Extraction" module in Figure 5-2 for both the processes. This indicates that no matter which document we want to classify and predict, it must go through the same series of transformations in the training and prediction process. Each document in first preprocessed and normalized and then specific features pertaining to the document are extracted. These processes are always uniform in both the "training" and "prediction" processes to make sure that our classification model performs consistently in its predictions.

In the *"training"* process, each document has its own corresponding class/category, which was manually labeled or curated beforehand. These training text documents are preprocessed and normalized in the "Text Normalization" module, giving us clean and standardized training text documents. They are then passed to the "Feature Extraction" module, where different feature extraction or engineering techniques are used to extract meaningful features from the processed text documents. Popular feature extraction techniques were covered extensively in Chapter 4 and we use some of them in this chapter! These features are usually numeric arrays or vectors, the reason being that standard machine learning algorithms work only on numeric vectors and can't work on raw unstructured data like text. Once we have our features, we select one or more than one supervised machine learning algorithms and train our model.

Training the model involves feeding the feature vectors from the documents and the corresponding labels such that the algorithm can learn various patterns corresponding to each class/category and can reuse this knowledge to predict classes for future new documents. Often an optional validation dataset is used to evaluate the performance of the classification algorithm to make sure it generalizes well with the data during training. A combination of these features and the machine learning algorithm yields a classification model, which is the end artifact or output from the "training" process. Often this model is tuned using various model parameters using a process called *hyperparameter* tuning to build a better performing optimal model. We explore this shortly during our hands-on examples.

The *"prediction"* process involves trying to either predict classes for new documents or evaluate how predictions are working on new, previously unseen, test data. The test dataset documents go through the same process of normalization and feature extraction and engineering. Then, the test document feature vectors are passed to the trained "classification model," which predicts the possible class for each of the documents based on previously learned patterns (no training happens here—maybe later if you have a model that learns from feedback). If you have the true class labels for the documents that were manually labeled, you can evaluate the prediction performance of the model by comparing the true labels and the predicted labels using various metrics like accuracy, precision, recall, and F1-score, to name a few. This would give you an idea of how well the model performs based on its predictions for new documents.

Once we have a stable and working model, the last step is to deploy the model, which usually involves saving the model and its necessary dependencies and deploying it as a service, API, or as a running program. It predicts categories for new documents as a batch job or based on serving user requests if accessed as a web service. There are various ways to deploy machine learning models and this usually depends on how you would want to access it later. We now discuss each of the main modules and components from this blueprint and implement/reuse these modules so that we can integrate them to build a real-world automated text classifier.

Data Retrieval

Obviously, the first step in any data science or machine learning pipeline is to access and retrieve the data necessary for our analysis and for building machine learning models. For this, we use the very popular but non-trivial 20 Newsgroups dataset, which is available for download directly using Scikit-Learn. The 20 Newsgroups dataset comprises around 18,000 newsgroups posts spread across 20 different categories or topics, thus making it a 20-class classification problem, which is definitely non-trivial as compared to predicting spam in emails. Remember, the higher the number of classes, the more complex it gets to build an accurate classifier.

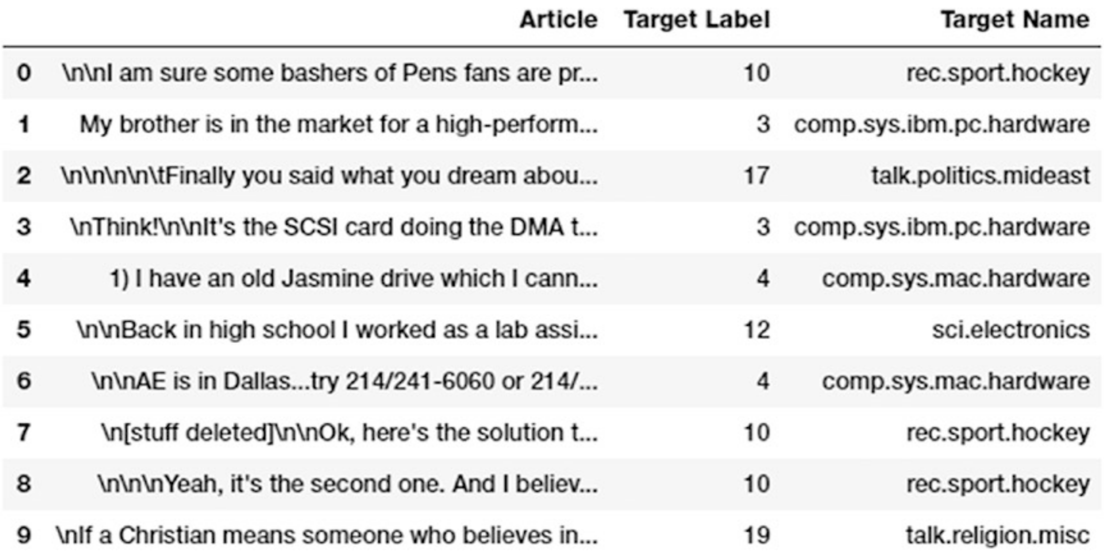
Details pertaining to the dataset can be found at `http://scikit-learn.org/0.19/datasets/twenty_newsgroups.html` and it is recommended to remove the headers, footers, and quotes from the text documents to prevent the model from overfitting or not generalizing well due to certain specific headers or email addresses. Thankfully, Scikit-Learn recognizes this problem and the functions that load the 20 Newsgroups data provide a parameter called `remove`, telling it what kinds of information to strip out of each file. The `remove` parameter should be a tuple containing any subset of `('headers', 'footers', 'quotes')`, telling it to remove headers, signature blocks, and quotation blocks, respectively.

We will also remove documents that are empty or have no content after removing these three items during the data preprocessing stage, because it would be pointless to try to extract features from empty documents. Let's start by loading the necessary dataset and defining functions for building the training and testing datasets. We load the usual dependencies including the text preprocessing and normalization module we built in Chapter 3, called `text_normalizer`.

```
from sklearn.datasets import fetch_20newsgroups
import numpy as np
import text_normalizer as tn
import matplotlib.pyplot as plt
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

We now leverage the helper function from `Scikit-Learn` to fetch the required data. Once we get the data, we transform this data into an easy-to-use dataframe. See Figure 5-3.

```
data = fetch_20newsgroups(subset='all', shuffle=True,
                          remove=('headers', 'footers', 'quotes'))
data_labels_map = dict(enumerate(data.target_names))

Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 MB)

# building the dataframe
corpus, target_labels, target_names = (data.data, data.target,
                                       [data_labels_map[label] for label in
data.target])
data_df = pd.DataFrame({'Article': corpus, 'Target Label': target_labels,
'Target Name': target_names})
print(data_df.shape)
data_df.head(10)

(18846, 3)
```



	Article	Target Label	Target Name
0	\n\nI am sure some bashers of Pens fans are pr...	10	rec.sport.hockey
1	My brother is in the market for a high-perform...	3	comp.sys.ibm.pc.hardware
2	\n\n\n\n\tFinally you said what you dream abou...	17	talk.politics.mideast
3	\nThink!\n\nIt's the SCSI card doing the DMA t...	3	comp.sys.ibm.pc.hardware
4	1) I have an old Jasmine drive which I cann...	4	comp.sys.mac.hardware
5	\n\nBack in high school I worked as a lab assi...	12	sci.electronics
6	\n\nAE is in Dallas...try 214/241-6060 or 214/...	4	comp.sys.mac.hardware
7	\n[stuff deleted]\n\nOk, here's the solution t...	10	rec.sport.hockey
8	\n\n\nYeah, it's the second one. And I believ...	10	rec.sport.hockey
9	\nIf a Christian means someone who believes in...	19	talk.religion.misc

Figure 5-3. *The 20 Newsgroups dataset*

From this dataset, we can see that each document has some textual content and the label can be denoted by a specific number, which maps to a newsgroup category name. Some data samples are depicted in Figure 5-3.

Data Preprocessing and Normalization

Before, we preprocess and normalize our documents, let's first take a look at potential empty documents in our dataset and remove them.

```
total_nulls = data_df[data_df.Article.str.strip() == "].shape[0]
print("Empty documents:", total_nulls)
```

```
Empty documents: 515
```

We can now do use a simple pandas filter operation and remove all the records with no textual content in the article as follows.

```
data_df = data_df[~(data_df.Article.str.strip() == ")]
data_df.shape
```

```
(18331, 3)
```

This is neat! Now we need to think about the general text preprocessing or wrangling stage. This involves cleaning, preprocessing, and normalizing text to bring text components like sentences, phrases, and words to some standard format. This enables standardization across our document corpus, which helps in building meaningful features and helps reduce noise, which can be introduced due to many factors like irrelevant symbols, special characters, XML and HTML tags, and so on. We have already talked about this in detail in Chapter 3. However, just for a brief recap, the main components in our text normalization pipeline are described as follows. Remember they are all available as a part of the `text_normalizer` module, which is present in the `text_normalizer.py` file.

- **Cleaning text:** Our text often contains unnecessary content, like HTML tags, which do not add much value when analyzing sentiment. Hence, we need to make sure we remove them before extracting features. The BeautifulSoup library does an excellent job at providing necessary functions for this. Our `strip_html_tags(...)` function cleans and strips out HTML code.

- **Removing accented characters:** In our dataset, we are dealing with reviews in the English language so we need to make sure that characters with any other format, especially accented characters, are converted and standardized into ASCII characters. A simple example is converting é to e. Our `remove_accented_chars(...)` function helps in this respect.

- **Expanding contractions:** In the English language, contractions are basically shortened versions of words or syllables. These shortened versions of existing words or phrases are created by removing specific letters and sounds. More often than not, vowels are removed from the words. Examples include do not to don't and I would to I'd. Contractions pose a problem in text normalization because we have to deal with special characters like apostrophes and we also have to convert each contraction to its expanded, original form. The `expand_contractions(...)` function uses regular expressions and various contractions mapped in the `contractions.py` module to expand all contractions in our text corpus.

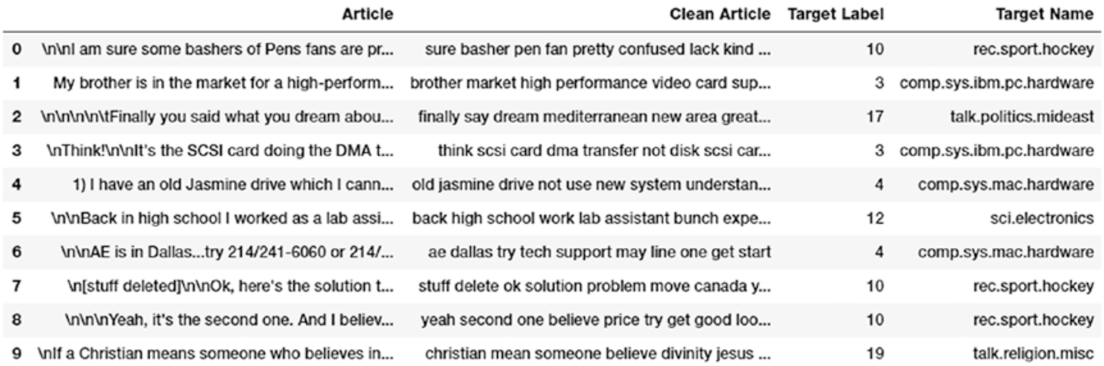
- **Removing special characters:** Another important task in text cleaning and normalization is to remove special characters and symbols that often add to the extra noise in unstructured text. Simple regexes can be used to achieve this. Our function `remove_special_characters(...)` helps remove special characters. In our code, we have retained numbers but you can also remove numbers if you do not want them in your normalized corpus.

- **Stemming or lemmatization:** Word stems are usually the base form of possible words that can be created by attaching affixes, like prefixes and suffixes, to the stem to create new words. This is known as *inflection*. The reverse process of obtaining the base form of a word is known as *stemming*. A simple example is "watches," "watching," and "watched," which have the word root stem "watch" as the base form. The NLTK package offers a wide range of stemmers like the `PorterStemmer` and `LancasterStemmer`. Lemmatization is very similar to stemming, where we remove word affixes to get to the base form of a word. However, the base form in this case is known as the root word but not the root stem. The difference being that the root word is always a lexicographically correct word (present in the dictionary) but the root stem may not be correct. We use lemmatization only in our normalization pipeline to retain lexicographically correct words. The function `lemmatize_text(...)` helps us in that respect.

- **Removing stopwords:** Words that have little or no significance, especially when constructing meaningful features from text, are known as stopwords. These are usually words that end up having the maximum frequency if you do a simple term or word frequency in a document corpus. Words like "a," "an," "the," and so on are stopwords. There is no universal stopword list but we use a standard English language stopwords list from NLTK. You can add your own domain specific stopwords if needed. The function `remove_stopwords(...)` helps remove stopwords and retain words having the most significance and context in a corpus.

We use all these components and tie them together in the function called `normalize_corpus(...)`, which can be used to take a document corpus as input and return the same corpus with cleaned and normalized text documents. This is already available in our text normalization module. Let's put this to the test now! See Figure 5-4.

```
import nltk
stopword_list = nltk.corpus.stopwords.words('english')
# just to keep negation if any in bi-grams
stopword_list.remove('no')
stopword_list.remove('not')

# normalize our corpus
norm_corpus = tn.normalize_corpus(corpus=data_df['Article'], html_stripping=True,
                                  contraction_expansion=True, accented_char_removal=True,
                                  text_lower_case=True, text_lemmatization=True,
                                  text_stemming=False, special_char_removal=True,
                                  remove_digits=True, stopword_removal=True,
                                  stopwords=stopword_list)
data_df['Clean Article'] = norm_corpus

# view sample data
data_df = data_df[['Article', 'Clean Article', 'Target Label', 'Target Name']]
data_df.head(10)
```



	Article	Clean Article	Target Label	Target Name
0	\n\nI am sure some bashers of Pens fans are pr...	sure basher pen fan pretty confused lack kind ...	10	rec.sport.hockey
1	My brother is in the market for a high-perform...	brother market high performance video card sup...	3	comp.sys.ibm.pc.hardware
2	\n\n\n\n\tFinally you said what you dream abou...	finally say dream mediterranean new area great...	17	talk.politics.mideast
3	\nThink!\n\nIt's the SCSI card doing the DMA t...	think scsi card dma transfer not disk scsi car...	3	comp.sys.ibm.pc.hardware
4	1) I have an old Jasmine drive which I cann...	old jasmine drive not use new system understan...	4	comp.sys.mac.hardware
5	\n\nBack in high school I worked as a lab assi...	back high school work lab assistant bunch expe...	12	sci.electronics
6	\n\nAE is in Dallas...try 214/241-6060 or 214/...	ae dallas try tech support may line one get start	4	comp.sys.mac.hardware
7	\n[stuff deleted]\n\nOk, here's the solution t...	stuff delete ok solution problem move canada y...	10	rec.sport.hockey
8	\n\n\nYeah, it's the second one. And I believ...	yeah second one believe price try get good loo...	10	rec.sport.hockey
9	\nIf a Christian means someone who believes in...	christian mean someone believe divinity jesus ...	19	talk.religion.misc

Figure 5-4. *The 20 Newsgroups dataset after text preprocessing*

We now have a nice preprocessed and normalized corpus of articles. But wait, it's not over yet! There might have been some documents that, after preprocessing, might end up being empty or null. We use the following code to test this assumption and remove these documents from our corpus.

```
data_df = data_df.replace(r'^(\s?)+$', np.nan, regex=True)
data_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 18331 entries, 0 to 18845
Data columns (total 4 columns):
Article          18331 non-null object
Clean Article    18304 non-null object
Target Label     18331 non-null int64
Target Name      18331 non-null object
dtypes: int64(1), object(3)
memory usage: 1.3+ MB
```

We definitely have some null articles after our preprocessing operation. We can safely remove these null documents using the following code.

```
data_df = data_df.dropna().reset_index(drop=True)
data_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18304 entries, 0 to 18303
Data columns (total 4 columns):
Article          18304 non-null object
Clean Article    18304 non-null object
Target Label     18304 non-null int64
Target Name      18304 non-null object
dtypes: int64(1), object(3)
memory usage: 572.1+ KB
```

We can now use this dataset for building our text classification system. Feel free to store the dataset using the following code if needed so you don't need to run the preprocessing step every time.

```
data_df.to_csv('clean_newsgroups.csv', index=False)
```

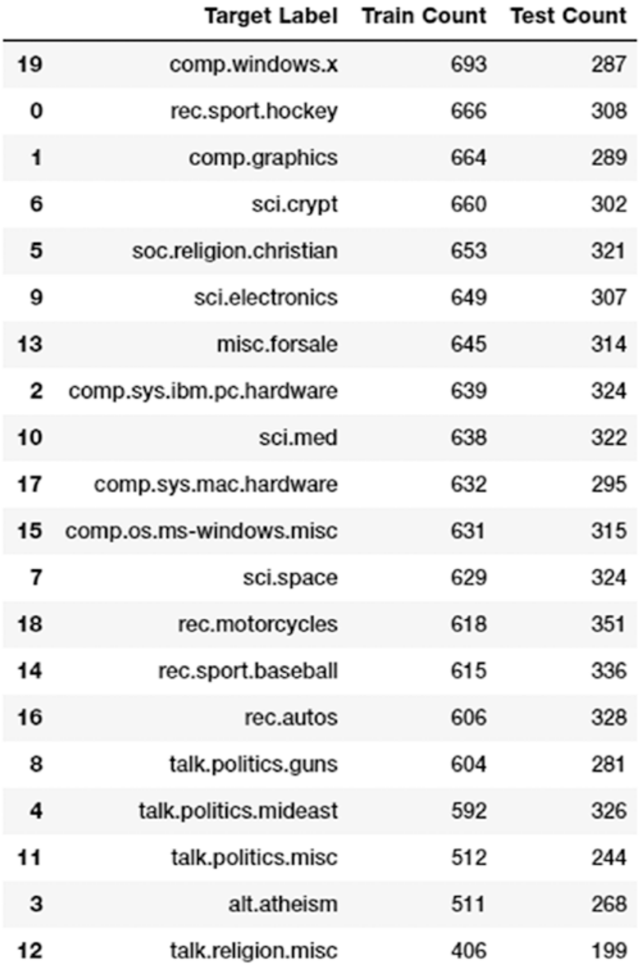
Building Train and Test Datasets

To build a machine learning system, we need to build our models on training data and then test and evaluate their performance on test data. Hence, we split our dataset into train and test datasets. We take a train dataset : test dataset split of 67%/33% of the total data.

```
from sklearn.model_selection import train_test_split

train_corpus, test_corpus, train_label_nums, test_label_nums, train_label_
names, test_label_names = train_test_split(np.array(data_df['Clean Article']),
                                           np.array(data_df['Target Label']),
                                           np.array(data_df['Target Name']),
                                           test_size=0.33, random_state=42)

train_corpus.shape, test_corpus.shape

((12263,), (6041,))
```

You can also observe the distribution of the various articles by the different newsgroup categories using the following code. We can then get an idea of how many documents will be used to train the model and how many are used to test the model. See Figure 5-5.

```
from collections import Counter

trd = dict(Counter(train_label_names))
tsd = dict(Counter(test_label_names))

(pd.DataFrame([[key, trd[key], tsd[key]] for key in trd],
             columns=['Target Label', 'Train Count', 'Test Count'])
.sort_values(by=['Train Count', 'Test Count'],
             ascending=False))
```



	Target Label	Train Count	Test Count
19	comp.windows.x	693	287
0	rec.sport.hockey	666	308
1	comp.graphics	664	289
6	sci.crypt	660	302
5	soc.religion.christian	653	321
9	sci.electronics	649	307
13	misc.forsale	645	314
2	comp.sys.ibm.pc.hardware	639	324
10	sci.med	638	322
17	comp.sys.mac.hardware	632	295
15	comp.os.ms-windows.misc	631	315
7	sci.space	629	324
18	rec.motorcycles	618	351
14	rec.sport.baseball	615	336
16	rec.autos	606	328
8	talk.politics.guns	604	281
4	talk.politics.mideast	592	326
11	talk.politics.misc	512	244
3	alt.atheism	511	268
12	talk.religion.misc	406	199

Figure 5-5. *Distribution of train and test articles by the 20 newsgroups*

We now briefly cover the various feature engineering techniques, which we use in this chapter to build our text classification models.

Feature Engineering Techniques

There are various feature extraction or feature engineering techniques that can be applied on text data, but before we jump into then, let’s briefly recap what we mean by features, why we need them, and how they are useful. In a dataset, there are typically many data points, which are usually the rows of the dataset, and the columns are various features or properties of the dataset with specific values for each row or observation.

In machine learning terminology, features are unique measurable attributes or properties for each observation or data point in a dataset. Features are usually numeric in nature and can be absolute numeric values or categorical features that can be encoded as binary features for each category in the list using a process called *one-hot encoding*. They can be represented as distinct numerical entities using a process called *label-encoding*. The process of extracting and selecting features is both an art and a science and this process is called *feature extraction* or *feature engineering*.

Feature engineering is very important and is often known as the secret sauce to creating superior and better performing machine learning models. Extracted features are fed into machine learning algorithms for learning patterns that can be applied on future new data points for getting insights. These algorithms usually expect features in the form of numeric vectors because each algorithm is at heart a mathematical operation of optimization and minimizing loss and error when it tries to learn patterns from data points and observations. Hence, with textual data comes the added challenge of figuring out how to transform and extract numeric features from textual data.

We covered state-of-the-art feature engineering techniques for text data in detail in Chapter 4. In the following sections we briefly recap the methods used in this chapter. But for a deep dive, I recommend readers check out Chapter 4.

Traditional Feature Engineering Models

Traditional (count-based) feature engineering strategies for textual data involve models belonging to a family of models, popularly known as the Bag of Words model in general. While they are effective methods for extracting features from text, due to the inherent nature of the model being just a bag of unstructured words, we lose additional information like the semantics, structure, sequence, and context around nearby words in each text document.

- **Bag of Words (term frequency) model**: The Bag of Words model represents each text document as a numeric vector where each dimension is a specific word from the corpus and the value could be its frequency in the document, occurrence (denoted by 1 or 0), or even weighted values. The model's name is such because each document is represented literally as a bag of its own words, disregarding word orders, sequences, and grammar.

- **Bag of N-Grams model:** An N-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence. Bi-grams indicate n-grams of order 2 (two words), tri-grams indicate n-grams of order 3 (three words), and so on. The Bag of N-Grams model is just an extension of the Bag of Words model so we can also leverage N-gram based features.
- **TF-IDF model:** TF-IDF stands for Term Frequency-Inverse Document Frequency and it's a combination of two metrics, term frequency (TF) and inverse document frequency (IDF). This technique was originally developed as a metric for for showing search engine results based on user queries and has become part of information retrieval and text feature extraction.

Advanced Feature Engineering Models

Traditional (count-based) feature engineering strategies for textual data involve models belonging to a family of models popularly known as the Bag of Words model. This includes term frequencies, TF-IDF (term frequency-inverse document frequency), N-grams, and so on. While they are effective methods for extracting features from text, there are severe limitations where we lose additional information like the semantics, structure, sequence, and context around nearby words in each text document. This forms as enough motivation for us to explore more sophisticated models that can capture this information and give us features that are vector representation of words, popularly known as *embeddings*. We use predictive methods, like Neural Network based Language Models, which try to predict words from its neighboring words by looking at word sequences in the corpus. In the process, it learns distributed representations giving us dense word embeddings. These models are commonly also known as word embedding models.

- **Word2Vec model:** This model was created by Google in 2013 and is a predictive deep learning based model to compute and generate high quality, distributed, and continuous dense vector representations of words that capture contextual and semantic similarity. You can usually specify the size of the word embedding vectors and the total number of vectors is essentially the size of the vocabulary. This makes the dimensionality of this dense vector space much lower than the high-dimensional sparse vector space built using traditional Bag of Words models. There are two different model architectures that can be leveraged by Word2Vec to create these word embedding representations. These include The Continuous Bag of Words (CBOW) model and the Skip-Gram model.

- **GloVe model:** The GloVe model stands for Global Vectors. It's an unsupervised learning model that can be used to obtain dense word vectors, similar to Word2Vec. However, the technique is different and training is performed on an aggregated global word-word co-occurrence matrix, giving us a vector space with meaningful sub-structures.
- **FastText model:** The FastText model was introduced by Facebook in 2016 as an extension and supposedly an improvement of the vanilla Word2Vec model. It's based on the original paper entitled "Enriching Word Vectors with Subword Information" by Mikolov et al., which is an excellent read to gain an in-depth understanding of how this model works. Overall, FastText is a framework for learning word representations and performing robust, fast, and accurate text classifications. The Word2Vec model typically ignores the morphological structure of each word and considers a word a single entity. The FastText model considers each word a Bag of Character n-grams. This is also called a *subword* model in the paper.

This should give us enough perspective into the types of feature engineering techniques that we use in our articles to get effective feature representation in the form of structured numeric vectors from unstructured textual data. In the next section, we take a quick conceptual glance at some of the common supervised learning/ classification models that we use later to build our text classification system.

Classification Models

Classification models are supervised machine learning algorithms that are used to classify, categorize, or label data points based on what it has observed in the past. Each classification algorithm is a supervised learning algorithm so it requires training data. This training data consists of a set of training observations where each observation is a pair that consists of an input data point, which is usually a feature vector like we observed earlier, and a corresponding output outcome for that input observation. There are three stages that classification algorithms go through during the modeling phase:

- Training
- Evaluation
- Tuning

Training is the process where the supervised learning algorithm tries to infer patterns out of the training data so that it can identify which patterns lead to a specific outcome. These outcomes are often known as the class labels/class variables/response variables. We usually carry out the process of feature extraction or feature engineering to derive meaningful features from the raw data before training. These feature sets are fed to an algorithm of our choice, which then tries to identify patterns from these features and their corresponding outcomes. The result of this is an inferred function known as a model or a classification model. This model is expected to be generalized enough from learning patterns in the training set so that it can predict the classes or outcomes for new data points in the future.

Evaluation involves trying to test the prediction performance of our model to see how well it has trained and learned on the training dataset. For this, we use a validation dataset and test the performance of our model by predicting on that dataset and testing our predictions against the actual class labels, also called the *ground truth.* We also often use cross validation where the data is divided into folds and a chunk of it is used for training and the remaining is used to validate the trained model.

A point to remember is that we also tune the model based on the validation results to get to an optimal configuration, which yields the maximum accuracy and minimum error. We also evaluate our model against a holdout or test dataset, but we never tune our model against that dataset because that would lead to the model being biased or overfit against very specific features from the test dataset. The holdout or test dataset is somewhat of a representative sample of what new real data samples might look like, for which the model will generate predictions and how it might perform on these new data samples. We look at various metrics, which are typically used to evaluate and measure model performance in a future section.

Tuning is also known as hyperparameter tuning or optimization, where we focus on trying to optimize a model to maximize its prediction power and reduce errors. Each model is at heart a mathematical function, which has several parameters determining model complexity, learning capability, and so on. These are known as *hyperparameters* because they cannot be learned directly from data and must be set prior to running and training the model. Hence, the process of choosing an optimal set of model hyperparameters such that the performance of the model yields good prediction accuracy is known as *model tuning* and we can carry it out in various ways, like randomized search and grid search. We look at some model tuning aspects during our hands-on implementations.

Typically, there are various classification algorithms but we will not be venturing into each algorithm in detail since the scope of this chapter is related to text classification and this is not a book only focusing on machine learning. However, we will touch upon a few algorithms, which we use shortly when building our classification models.

- Multinomial Naïve Bayes
- Logistic regression
- Support vector machines
- Random forest
- Gradient boosting machine

There are also several other classification algorithms; however, these are some of the most common and popular algorithms for text data. The last two models mentioned in this list are ensemble techniques, which use a collection or ensemble of models to learn and predict outcomes, including random forests and gradient boosting. Besides these, deep learning based techniques have also recently become popular which use multiple hidden layers and combine several neural network models to build a complex classification model. We now briefly look at some basic concepts surrounding these algorithms before using them for our classification problem.

Multinomial Naïve Bayes

This is a special case of the popular Naïve Bayes algorithm used specifically for prediction and classification tasks where we have more than two classes. Before looking at multinomial Naïve Bayes, let's look at the definition and formulation of the Naïve Bayes algorithm. The Naïve Bayes algorithm is a supervised learning algorithm that puts into action the very popular Bayes theorem. However there is a *"naïve"* assumption here that each feature is completely independent of the others. Mathematically, we can formulate this as, given a response class variable y and a set of **n** features in the form of a feature vector, $\{x_1, x_2, \ldots, x_n\}$ and using the Bayes theorem, we can denote the probability of the occurrence of y given the features as follows:

$$P(y|x_1, x_2, \ldots, x_n) = \frac{P(y) \times P(x_1, x_2, \ldots, x_n|y)}{P(x_1, x_2, \ldots, x_n)}$$

under the assumption that

$$P(x_i|y, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y)$$

and for all i we can represent this as follows:

$$P(y|x_1, x_2, \ldots, x_n) = \frac{P(y) \times \prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, \ldots, x_n)}$$

where i ranges from 1 to **n**. In simple terms, this can be written as follows:

$$posterior = \frac{prior \times likelihood}{evidence}$$

and now since $P(x_1, x_2, \ldots, x_n)$ is constant, the model can be expressed as follows:

$$P(y|x_1, x_2, \ldots, x_n) \; \alpha \; P(y) \times \prod_{i=1}^{n} P(x_i|y)$$

This means that under the previous assumptions of independence among the features where each feature is conditionally independent of every other feature, the conditional distribution over the class variable to be predicted, y, can be represented using the following mathematical equation as follows:

$$P(y|x_1, x_2, \ldots, x_n) = \frac{1}{Z} P(y) \times \prod_{i=1}^{n} P(x_i|y)$$

where the evidence measure, $Z = p(x)$, is a constant scaling factor dependent on the feature variables. From this equation, we can build the Naïve Bayes classifier by combining it with a rule known as the MAP decision rule, which stands for maximum a posteriori. Going into the statistical details would be impossible in the current scope but by using it, the classifier can be represented as a mathematical function, which can assign a predicted class label $\hat{y} = C_k$ for some k using the following representation:

$$\hat{y} = \underset{k \in \{1, 2, \ldots, K\}}{\operatorname{argmax}} P(C_k) \times \prod_{i=1}^{n} P(x_i | C_k)$$

This classifier is often said to be simple, quite evident from its name and because of several assumptions that we make about our data and features that might not be so in the real world. Nevertheless, this algorithm still works remarkably well in many use cases related to classification, including multi-class document classification, spam filtering, and so on. They can train really fast compared to other classifiers and work well even when we do not have sufficient training data. Models often do not perform well when they have a lot of features and this phenomenon is known as curse of dimensionality. Naïve Bayes takes care of this problem by decoupling the class variable related conditional feature distributions, thus leading to each distribution being independently estimated as a single dimension distribution.

Multinomial Naïve Bayes is an extension of the algorithm for predicting and classifying data points, where the number of distinct classes or outcomes are more than two. In this case, the feature vectors are usually assumed to be word counts from the bag of words model, but TF-IDF-based weights also work. One limitation is that negative weight based features can't be fed into this algorithm. This distribution can be represented as $p_y = \{p_{y1}, p_{y2}, \ldots, p_{yn}\}$ for each class label y and the total number of features is n which could be represented as the total vocabulary of distinct words or terms in text analytics. From the equation, $p_{yi} = P(x_i|y)$ represents the probability of feature i in any observation sample that has an outcome or class y. The parameter p_y can be estimated with a smoothened version of maximum likelihood estimation (with relative frequency of occurrences) and represented as follows:

$$\hat{p}_{yi} = \frac{F_{yi} + \alpha}{F_y + \alpha n}$$

where $F_{yi} = \sum_{x \in TD} x_i$ is the frequency of occurrence for the feature i in a sample for class label y in our training dataset TD and $F_y = \sum_{i=1}^{|TD|} F_{yi}$ is the total frequency of all features for the class label y. There is some amount of smoothening done with the help of priors

$\alpha \geq 0$, which accounts for the features that are not present in the learning data points and helps get rid of zero probability related issues. There are some specific settings for this parameter, which are used quite often. The value of $\alpha = 1$ is known as *Laplace smoothing* and $\alpha < 1$ is known as *Lidstone smoothing*. The Scikit-Learn library provides an excellent implementation for Multinomial Naïve Bayes in the class `MultinomialNB`, which we leverage when we build our text classifier later. Remember not to set the α value to be too high blindly because this can lead the model to assume wrongly that some features that are not present are important features for predicting specific classes due to excessive smoothing.

Logistic Regression

The logistic regression model is actually a statistical model developed by statistician David Cox in 1958. It is also known as the logit or logistic model since it uses the logistic (popularly also known as sigmoid) mathematical function to estimate the parameter values. These are the coefficients of all our features such that the overall loss is minimized when predicting the outcome—in this case, the newsgroup categories. However, we don't focus on errors but more about maximizing the likelihood of the predicted values to the observed values using Maximum-Likelihood Estimation (MLE).

Considering a binary classification problem of predicting two classes, a 0 or a 1, in the logistic model, the log-odds (the logarithm of the odds) for the class/category labeled as 1 are basically the equation of the linear regression model (linear combination of one or more independent features, which can be categorical or continuous). However, we need to predict discrete classes or categories. Thus, the corresponding probability of the class labeled 1 can vary between 0 and 1, depicting the confidence of the prediction. The function that helps us convert the log-odds to probability is the logistic function. The standard sigmoid or logistic function can be depicted mathematically by this formula:

$$\frac{1}{1+e^{-x}}$$

Where e is the exponent (Euler's number) and x indicates the typical equation, which can be derived from the linear regression equation where we try to estimate the coefficients of our features. This function typically looks like an S-shaped curve, as depicted in Figure 5-6.

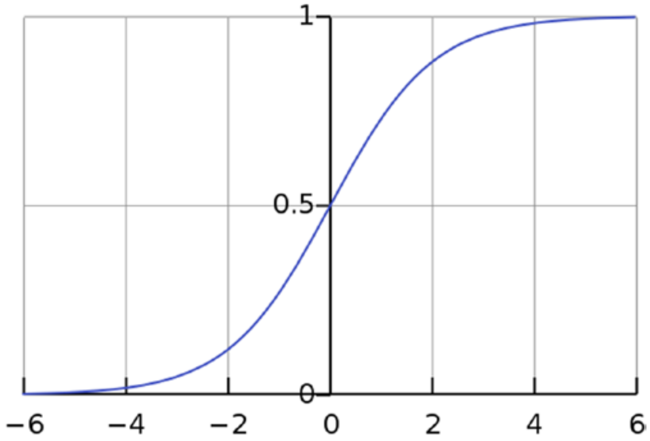


Figure 5-6. *The standard sigmoid or logistic function*

The standard unit of measurement for the log-odds scale is called a *logit*, from logistic unit, hence we have the alternative names for this model. To understand how this model works, you need to dive into the math and the intent of this book is not to give a course in machine learning. However, we briefly cover this for our more math-oriented folks! Consider a standard multiple linear regression model, depicted as follows:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

Such that $\{x_1, x_2, \ldots, x_n\}$ are our features and we are trying to estimate the coefficients, $\{\beta_1, \beta_2, \ldots, \beta_n\}$. Considering we need to predict the categorical classes, we can represent this as the log-odds, as follows using the *logit* of the probability *p*.

$$logodds = logit(p) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

This means that if *p* is the probability of predicting a specific class, the odds of that is $\frac{p}{1-p}$, which is basically the ratio of the favorable outcomes to the unfavorable outcomes. Likewise, the logit of *p* is basically the log-odds. Thus, we can mathematically derive this as follows:

$$logit(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

If we want to get to the class probability values that the logistic regression model outputs for us, we can derive the following equation, which is the heart of the logistic regression model:

$$p = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\cdots+\beta_n x_n)}}$$

Finally, we can use MLE to optimize and estimate the optimal coefficients for each feature, which helps in maximizing the likelihood function. In Scikit-Learn, the `LogisticRegression` model can be leveraged to use the logistic regression model for classification. The solvers implemented in the `LogisticRegression` class are `"liblinear"`, `"newton-cg"`, `"lbfgs"`, `"sag"`, and `"saga"`. Each of them has its own distinct implementations. In the case of multi-class classification, just like in our problem, the training algorithm uses the one-vs-rest (OvR) scheme if the `multi_class` option is set to `ovr` and uses the cross-entropy loss if the `multi_class` option is set to `multinomial`.

Support Vector Machines

In machine learning, support vector machines, known popularly as SVMs, are supervised learning algorithms. They are used for classification, regression, novelty and anomaly, and outlier detection. Considering a binary classification problem, if we have training data such that each data point or observation belongs to a specific class, the SVM algorithm can be trained based on this data such that it can assign future data points into one of the two classes. This algorithm represents the training data samples as points in space such that points belonging to either class can be separated by a wide gap between them (hyperplane) and the new data points to be predicted are assigned classes based on which side of this hyperplane they fall into. This process is for a typical linear classification process. However, SVM can also perform non-linear classification by an interesting approach known as a *kernel trick,* where kernel functions are used to operate on high-dimensional feature spaces that are non-linear separable. Usually, inner products between data points in the feature space help achieve this.

The SVM algorithm takes in a set of training data points and constructs a hyperplane of a collection of hyperplanes for a high-dimensional feature space. The larger the margins of the hyperplane, the better the separation. This leads to lower generalization errors of the classifier. Let's represent this formally and mathematically. Considering a training dataset of **n** data points $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ such that the class variable $y_i \in \{-1, 1\}$ where each value indicates the class corresponding to the point $\vec{x}_i$. Each data point $\vec{x}_i$ is a feature vector. The objective of the SVM algorithm is to find the max-margin hyperplane which separates the set of data points having class label of $y_i = 1$ from the set of data points having class label $y_i = -1$ so that the distance between the hyperplane and sample data points from either class nearest to it is maximized. These sample data points are known as the *support vectors*. Figure 5-7 shows how the vector space with the hyperplane looks.

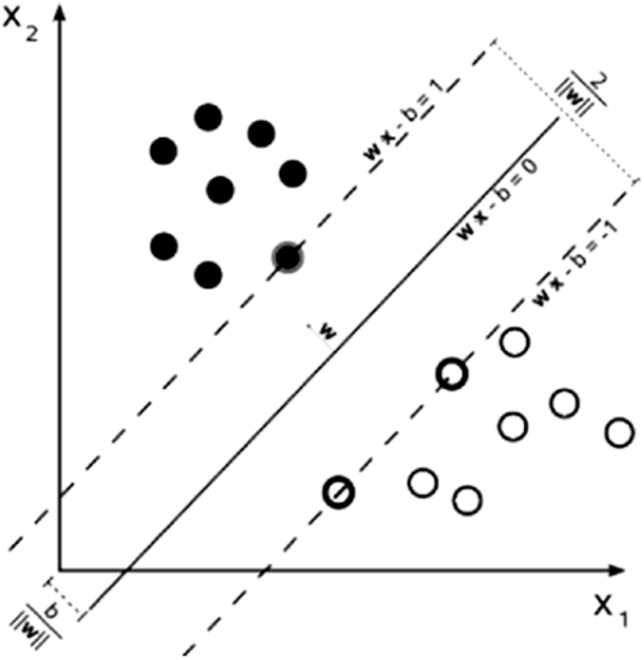


Figure 5-7. *Two-class SVM depicting hyperplane and support vectors*

From Figure 5-7, you can clearly see the hyperplane and the support vectors. The hyperplane can be defined as the set of points $\vec{x}$ which satisfy $\vec{w} \cdot \vec{x} + b = 0$, where $\vec{w}$ is the normal vector to the hyperplane and $\frac{b}{\|\vec{w}\|}$ gives us the offset of the hyperplane from the origin to the support vectors highlighted in Figure 5-7. There are two main types of margins that help in separating the data points belonging to the different classes.

When the data is linearly separable, like in Figure 5-7, we can have hard margins, which are basically represented by the two parallel hyperplanes depicted by the dotted lines. This helps in separating the data points belonging to the two different classes. This is done by taking into account that the distance between them is as large as possible. The region bounded by these two hyperplanes forms the margin with the max-margin hyperplane being in the middle. These hyperplanes have the equations $\vec{w} \cdot \vec{x} + b = 1$ and $\vec{w} \cdot \vec{x} + b = -1$.

Often, the data points are not linearly separable, for which we can use the hinge loss function, which can be represented as $\max\left(0, 1 - y_i\left(\vec{w} \cdot \vec{x}_i + b\right)\right)$. In fact, the Scikit-Learn implementation of SVM can be found in `SVC`, `LinearSVC`, or `SGDClassifier`, where we use the `hinge` loss function (set by default) to optimize and build the model. This loss function helps us get the soft margins and is often known as a *soft-margin SVM*. You can also use different kernel functions to convert the existing feature space into an even higher dimensional feature space, where the data can be separated linearly. This is popularly known as the *kernel trick* in SVM! However, we don't recommend this a lot for text data problems since you already deal with a huge number of dimensions right from the start.

For a multi-class classification problem, if we have **n** classes, for each class a binary classifier is trained and learned that helps is separating between each class and the other *n-1* classes. During prediction, the scores (distances to hyperplanes) for each classifier are computed and the maximum score is chosen for selecting the class label. The stochastic gradient descent is often used for minimizing the loss function in SVM algorithms. Figure 5-8 shows how three classifiers are trained in total for a three-class SVM problem over the very popular iris dataset. This figure is built using a Scikit-Learn model and is obtained from their official documentation, available at `http://scikit-learn.org/`.

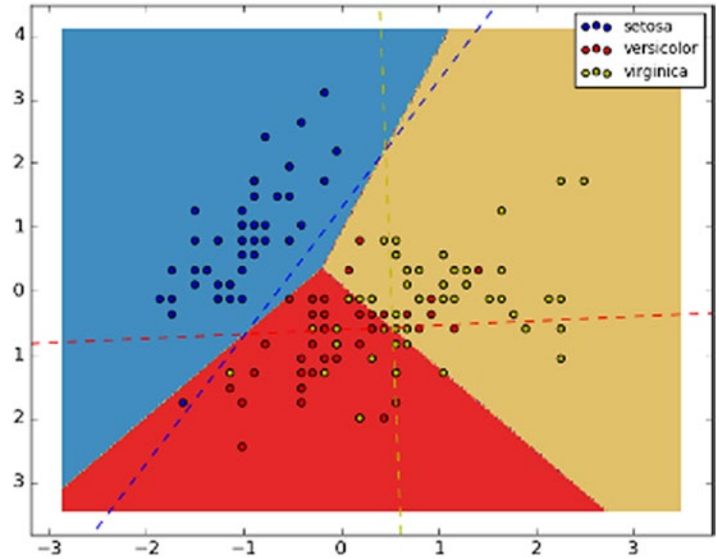


Figure 5-8. *Multi-class SVM on three classes (courtesy: scikit-learn.org)*

From Figure 5-8, you can clearly see that a total of three SVM classifiers have been trained for each of the three classes. They are combined for the final predictions so that data points belonging to each class can be labeled correctly. Thus, multi-class support is handled according to a one-vs-the-rest scheme, similar to the logistic regression model.

Ensemble Models

Ensemble models are essentially models or meta-estimators that are literally *made up of other models* or *estimators*. These sub-models are models that are simple estimators and may not be able to make accurate predictions to the extent of what you get when you combine several of these estimators. In the case of random forest, the sub-models are decision trees. Typically, random forests train many decision trees and combine them to generate a single prediction. There are a wide variety of ensemble models. We briefly mention two categories since we will be covering an example from each of these categories shortly.

- **Bagging**: A very popular ensemble modeling technique. In bagging, you take subsets of the data (bootstrap samples typically) and train a model on each subset in parallel. Then the subsets are allowed to *simultaneously* vote on the outcome and the final outcome is usually an average aggregation. The random forest model is perhaps the most popular example of a bagging model.

- **Boosting**: Another ensemble technique where, rather than building multiple models in parallel like with bagging, you use sequential modeling and try to improve one model from the mistakes of the previous model! Boosting typically uses the output of one model as an input into the next in a form of *sequential* processing. Gradient boosting machines is one of the most popular boosting models.

Random Forest

Decision trees are a family of supervised machine learning algorithms that can represent and interpret sets of rules automatically from the underlying data. They use metrics like information gain and gini-index to build the tree. However, a major drawback of decision trees is that since they are non-parametric, the more data there is, greater the depth of the tree. We can end up with really huge and deep trees that are prone to overfitting. The model might work really well on training data, but instead of learning, it just memorizes all the training samples and builds very specific rules to them. Hence, it performs really poorly on the test data. Random forests try to tackle this problem.

A random forest is a meta-estimator or an ensemble model that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size, but the samples are drawn with replacement (bootstrap samples). In random forests, all the trees are trained in parallel (bagging model/bootstrap aggregation). Besides this, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. Also, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. Thus the randomness introduced in a random forest is both due to random sampling of data and random selection of features when splitting nodes in each tree. Hence, due to this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random decision tree). However, due to averaging, the overall variance of the model decreases significantly as compared to the increase in bias and hence it gives us an overall better model.

When building a random forest, you can set specific model parameters for both the base decision trees and the overall forest. For the trees, you usually have the same parameters as a normal decision tree model like the tree depth, number of leaves, number of features in each split, samples per leaf, criteria for the node splits, information gain, and gini impurity. For the forest, you can tune the total number of trees needed, the number of features to be used per tree, and so on.

Gradient Boosting Machines

Gradient boosting machines, popularly known as GBMs, can be used for regression and classification. Typically, GBMs builds an additive model in a forward stage-wise sequential fashion; they allow for the optimization of arbitrary differentiable loss functions. GBMs can usually work on any combination of models (weak learners) and loss functions. Scikit-Learn uses GBRTs (Gradient Boosted Regression Trees), which are generalized boosting models that can be applied to arbitrary differentiable loss functions. The beauty of this model is that is accurate and can be used for both regression and classification problems. GBRT considers additive models that can be mathematically represented as follows:

$$F(x) = \sum_{m=1}^{M} \gamma_m h_m(x)$$

Where $h_m(x)$ can be defined as the base models or weak learners—in this case the decision trees. Similar to other boosting algorithms, GBRT builds the additive model in forward stage wise sequential manner. Mathematically, this can be represented as follows:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

At each stage, the next decision tree $h_m(x)$ is chosen to minimize the loss function L given the previous decision tree model F_{m-1} and its fit $F_{m-1}(x_i)$. This can be represented as follows:

$$F_m(x) = F_{m-1}(x) + \arg\min_{h} \sum_{i=1}^{n} L\big(y_i, F_{m-1}(x_i) + h(x)\big)$$

Typically, decision trees are used as the base models and we end up minimizing the residuals (regression trees) or the negative log likelihood (classification trees).

There are a lot of resources and books dedicated entirely to supervised machine learning and classification. We encourage readers to check them out to gain more in-depth knowledge into how these techniques work and how they can be applied to various problems in analytics. We also recommend readers check out the latest state-of-the-art ensemble models, like XGBoost, CatBoost, and LightGBM.

Evaluating Classification Models

Training, tuning, and building models are an important part of the whole analytics lifecycle, but it's even more important to know how well these models are performing. Performance of classification models is usually based on how well they are predicting outcomes for new data points. Usually this performance is measured against a test or holdout dataset, which consists of data points that were not used to influence or train the classifier in any way. This test dataset has several observations and their corresponding labels. We extract features in the same way as when training the model. These features are fed to the already trained model and we obtain predictions for each data point. These predictions are then matched against the actual labels to see how well or how accurately the model has predicted. There are several metrics to determine a model's prediction performance. We mainly focus on the following metrics.

- Accuracy
- Precision
- Recall
- F1-score

Let's take a classic example of the very popular Wisconsin Diagnostic Breast Cancer dataset. This dataset has 30 attributes or features and a corresponding label for each data point (breast mass) depicting if it has cancer (malignant: label value 1) or no cancer (benign: label value 0). Let's assume we already have this data and will be building a basic logistic regression model and evaluating it. We take the assumption that we have 398 observations in our train dataset and 171 observations in our test dataset. We will be leveraging a nifty module we have created for model evaluation. It is named `model_evaluation_utils` and you can find it along with the code files and notebooks for this chapter. We recommend you check out the code, which leverages the Scikit-Learn `metrics` module to compute most of the evaluation metrics and plots.

Confusion Matrix

A *confusion matrix* is one of the most popular ways to evaluate a classification model. Although the matrix by itself is not a metric, the matrix representation can be used to define a variety of metrics, all of which become important in some specific case or scenario. A confusion matrix can be created for both a binary classification as well as a multi-class classification model.

A confusion matrix is created by comparing the predicted class label of a data point with its actual class label. This comparison is repeated for the whole dataset and the results of this comparison are compiled in a matrix or tabular format. This resultant matrix is our confusion matrix. Before we go any further, let's build a logistic regression model on our breast cancer dataset and look at the confusion matrix for the model predictions on the test dataset.

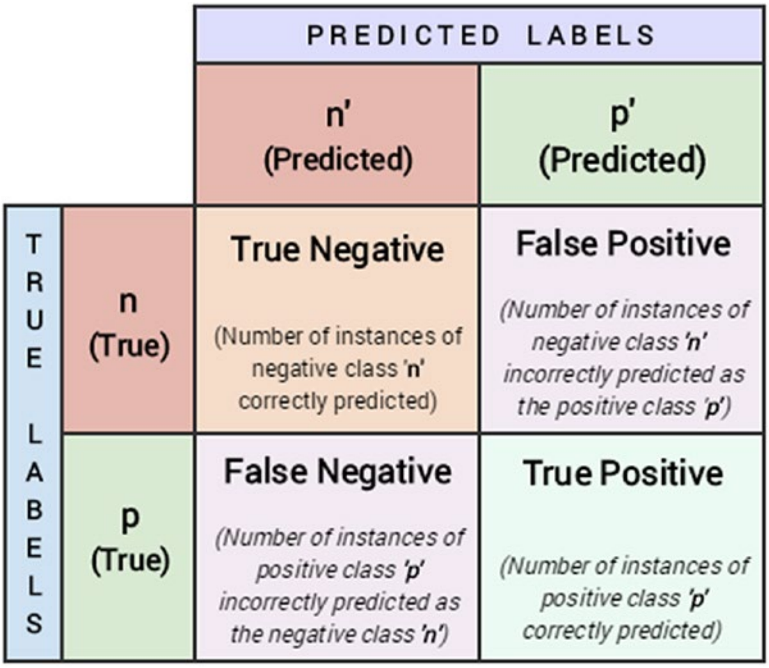
```
from sklearn import linear_model
# train and build the model
logistic = linear_model.LogisticRegression()
logistic.fit(X_train,y_train)

# predict on test data and view confusion matrix
import model_evaluation_utils as meu

y_pred = logistic.predict(X_test)
meu.display_confusion_matrix(true_labels=y_test, predicted_labels=y_pred,
classes=[0, 1])
```

```
          Predicted:
                   0    1
Actual: 0         59    4
        1          2  106
```

The preceding output depicts the confusion matrix with necessary annotations. We can see that out of 63 observations with label 0 (benign), our model has correctly predicted 59 observations. Similarly, out of 108 observations with label 1 (malignant), our model has correctly predicted 106 observations. More detailed analysis is coming right up!

Understanding the Confusion Matrix

While the name itself sounds pretty overwhelming, understanding the confusion matrix is not that confusing once you have the basics right! To reiterate what we learned in the previous section, the confusion matrix is a tabular structure that keeps a track of correct classifications as well as misclassifications. This is useful to evaluate the performance of a classification model for which we know the true data labels and can compare with the predicted data labels. Each column in the confusion matrix represents classified instance counts based on predictions from the model and each row of the matrix represents instance counts based on the actual/true class labels. This structure can also be reversed, i.e. predictions depicted by rows and true labels by columns. In a typical binary classification problem, we usually have a class label that's defined as the positive class, which is basically the class of our interest. For instance, in our breast cancer dataset, we are interested in detecting breast cancer, hence label 1 is our positive class. Figure 5-9 shows a typical confusion matrix for a binary classification problem, where p denotes the positive class and n denotes the negative class.



		PREDICTED LABELS	
		n' (Predicted)	p' (Predicted)
TRUE LABELS	n (True)	True Negative (Number of instances of negative class 'n' correctly predicted)	False Positive *(Number of instances of negative class 'n' incorrectly predicted as the positive class 'p')*
	p (True)	False Negative *(Number of instances of positive class 'p' incorrectly predicted as the negative class 'n')*	True Positive *(Number of instances of positive class 'p' correctly predicted)*

Figure 5-9. *Typical structure of a confusion matrix*

Figure 5-9 should make things more clear with regard to the structure of confusion matrices. In general, we usually have a positive class as we discussed earlier and the other class is the negative class. Based on this structure, we can clearly see four terms of importance.

- **True Positive (TP):** This is the count of the total number of instances from the positive class where the true class label was equal to the predicted class label, i.e., the total instances where we correctly predicted the positive class label with our model.
- **False Positive (FP):** This is the count of the total number of instances from the negative class where our model misclassified them by predicting them as positive. Hence, the name, "false" positive.
- **True Negative (FN):** This is the count of the total number of instances from the negative class, where the true class label was equal to the predicted class label, i.e., the total instances where we correctly predicted the negative class label with our model.
- **False Negative (FN):** This is the count of the total number of instances from the positive class where our model misclassified them by predicting them as negative. Hence the name, "false" negative.

Based on this information, can you compute these metrics for our confusion matrix based on the model predictions on the breast cancer test data?

```
positive_class = 1
TP = 106
FP = 4
TN = 59
FN = 2
```

Performance Metrics

The confusion matrix by itself is not a performance measure for classification models. But it can be used to calculate several metrics that are useful measures for different scenarios. We describe how the major metrics can be calculated from the confusion matrix, compute them manually using necessary formulae, compare the results with functions provided by Scikit-Learn on our predicted results, and give an intuition of scenarios where each of those metric can be used.

Accuracy is one of the most popular measures of classifier performance. It is defined as the overall proportion of correct predictions of the model. The formula for computing accuracy from the confusion matrix is as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Accuracy is normally used when our classes are almost balanced and correct predictions of those classes are equally important. The following code computes accuracy on our model predictions.

```
fw_acc = round(meu.metrics.accuracy_score(y_true=y_test, y_pred=y_pred), 5)
mc_acc = round((TP + TN) / (TP + TN + FP + FN), 5)
print('Framework Accuracy:', fw_acc)
print('Manually Computed Accuracy:', mc_acc)
```

```
Framework Accuracy: 0.96491
Manually Computed Accuracy: 0.96491
```

Precision, also known as positive predictive value, is another metric that can be derived from the confusion matrix. It is defined as the number of predictions made that are actually correct or relevant out of all the predictions based on the positive class. The formula for precision is as follows:

$$Precision = \frac{TP}{TP + FP}$$

A model with high precision will identify a higher fraction of positive classes as compared to a model with a lower precision. Precision becomes important in cases where we are more concerned about finding the maximum number of positive classes even if the total accuracy reduces. The following code computes precision on our model predictions.

```
fw_prec = round(meu.metrics.precision_score(y_true=y_test, y_pred=y_pred), 5)
mc_prec = round((TP) / (TP + FP), 5)
print('Framework Precision:', fw_prec)
print('Manually Computed Precision:', mc_prec)
```

```
Framework Precision: 0.96364
Manually Computed Precision: 0.96364
```

Recall, also known as *sensitivity*, is a measure of a model to identify the percentage of relevant data points. It is defined as the number of instances of the positive class that were correctly predicted. This is also known as hit rate, coverage, or sensitivity. The formula for recall is as follows:

$$Recall = \frac{TP}{TP + FN}$$

Recall becomes an important measure of classifier performance when we want to catch the most number of instances of a particular class even when it increases our false positives. For example, consider the case of bank fraud. A model with high recall will give us higher number of potential fraud cases. But it will also help us raise alarm for most of the suspicious cases. The following code computes recall on our model predictions.

```
fw_rec = round(meu.metrics.recall_score(y_true=y_test, y_pred=y_pred), 5)
mc_rec = round((TP) / (TP + FN), 5)
print('Framework Recall:', fw_rec)
print('Manually Computed Recall:', mc_rec)
```

```
Framework Recall: 0.98148
Manually Computed Recall: 0.98148
```

There are some cases in which we want a balanced optimization of both precision and recall. The F1-score is the harmonic mean of precision and recall and helps us optimize a classifier for balanced precision and recall performance.

The formula for the F1-score is as follows:

$$F1\, Score = \frac{2 \times \; Precision \times Recall}{Precision + Recall}$$

Let's compute the F1-score on the predictions made by our model using the following code.

```
fw_f1 = round(meu.metrics.f1_score(y_true=y_test, y_pred=y_pred), 5)
mc_f1 = round((2*mc_prec*mc_rec) / (mc_prec+mc_rec), 5)
print('Framework F1-Score:', fw_f1)
print('Manually Computed F1-Score:', mc_f1)
```

```
Framework F1-Score: 0.97248
Manually Computed F1-Score: 0.97248
```

Thus, you can see how our manually computed metrics match the results obtained from Scikit-Learn functions. This should give you a good idea of how to evaluate classification models with these metrics.

Building and Evaluating Our Text Classifier

We have gone through all the steps necessary for building a classification system, including data retrieval, wrangling, text preprocessing and normalization, feature extraction and engineering, classification models, and model performance evaluation. In this section, we put everything together to build and evaluate our text classification system! Our training and test datasets are cleaned and ready to go. We will use the following workflows to build our text classifiers.

- Traditional feature representation (BOW, TF-IDF) and classification models
- Advanced feature representation (Word2Vec, GloVe, FastText) and classification models

We also use techniques like cross-validation and grid search for evaluating as well as tuning for our best models.

Bag of Words Features with Classification Models

Let's start by using a basic Bag of Words, the term frequency-based feature engineering model, to extract features from our train and test datasets.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import cross_val_score

# build BOW features on train articles
cv = CountVectorizer(binary=False, min_df=0.0, max_df=1.0)
cv_train_features = cv.fit_transform(train_corpus)

# transform test articles into features
cv_test_features = cv.transform(test_corpus)
```

```
print('BOW model:> Train features shape:', cv_train_features.shape,
      ' Test features shape:', cv_test_features.shape)
```

```
BOW model:> Train features shape: (12263, 66865)  Test features shape:
(6041, 66865)
```

We now build several classifiers on these features using the training data and test their performance on the test dataset using all the classification models we discussed earlier. We also check model accuracies using five-fold cross validation just to see if the model performs consistently across the validation folds of data (we use this same strategy to tune the models later).

```
# Naïve Bayes Classifier
from sklearn.naive_bayes import MultinomialNB

mnb = MultinomialNB(alpha=1)
mnb.fit(cv_train_features, train_label_names)
mnb_bow_cv_scores = cross_val_score(mnb, cv_train_features, train_label_
names, cv=5)
mnb_bow_cv_mean_score = np.mean(mnb_bow_cv_scores)
print('CV Accuracy (5-fold):', mnb_bow_cv_scores)
print('Mean CV Accuracy:', mnb_bow_cv_mean_score)
mnb_bow_test_score = mnb.score(cv_test_features, test_label_names)
print('Test Accuracy:', mnb_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.68468102  0.68241042  0.67835304  0.67741935
0.6792144 ]
Mean CV Accuracy: 0.680415648396
Test Accuracy: 0.680185399768
```

```
# Logistic Regression
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='l2', max_iter=100, C=1, random_state=42)
lr.fit(cv_train_features, train_label_names)
lr_bow_cv_scores = cross_val_score(lr, cv_train_features, train_label_
names, cv=5)
lr_bow_cv_mean_score = np.mean(lr_bow_cv_scores)
print('CV Accuracy (5-fold):', lr_bow_cv_scores)
print('Mean CV Accuracy:', lr_bow_cv_mean_score)
lr_bow_test_score = lr.score(cv_test_features, test_label_names)
print('Test Accuracy:', lr_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.70418529  0.69788274  0.69384427  0.6998775
0.69599018]
Mean CV Accuracy: 0.698355996012
Test Accuracy: 0.703856977322
```

```
# Support Vector Machines
from sklearn.svm import LinearSVC

svm = LinearSVC(penalty='l2', C=1, random_state=42)
svm.fit(cv_train_features, train_label_names)
svm_bow_cv_scores = cross_val_score(svm, cv_train_features, train_label_
names, cv=5)
svm_bow_cv_mean_score = np.mean(svm_bow_cv_scores)
print('CV Accuracy (5-fold):', svm_bow_cv_scores)
print('Mean CV Accuracy:', svm_bow_cv_mean_score)
svm_bow_test_score = svm.score(cv_test_features, test_label_names)
print('Test Accuracy:', svm_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.64120276  0.64169381  0.64900122  0.64107799
0.63993453]
Mean CV Accuracy: 0.642582064348
Test Accuracy: 0.656679357722
```

```
# SVM with Stochastic Gradient Descent
from sklearn.linear_model import SGDClassifier

svm_sgd = SGDClassifier(loss='hinge', penalty='l2', max_iter=5, random_
state=42)
svm_sgd.fit(cv_train_features, train_label_names)
svmsgd_bow_cv_scores = cross_val_score(svm_sgd, cv_train_features, train_
label_names, cv=5)
svmsgd_bow_cv_mean_score = np.mean(svmsgd_bow_cv_scores)
print('CV Accuracy (5-fold):', svmsgd_bow_cv_scores)
print('Mean CV Accuracy:', svmsgd_bow_cv_mean_score)
```

```
svmsgd_bow_test_score = svm_sgd.score(cv_test_features, test_label_names)
print('Test Accuracy:', svmsgd_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.66030069  0.62459283  0.65185487  0.63209473
0.64157119]
Mean CV Accuracy: 0.642082864709
Test Accuracy: 0.633007780169
```

```
# Random Forest
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(n_estimators=10, random_state=42)
rfc.fit(cv_train_features, train_label_names)
rfc_bow_cv_scores = cross_val_score(rfc, cv_train_features, train_label_
names, cv=5)
rfc_bow_cv_mean_score = np.mean(rfc_bow_cv_scores)
print('CV Accuracy (5-fold):', rfc_bow_cv_scores)
print('Mean CV Accuracy:', rfc_bow_cv_mean_score)
rfc_bow_test_score = rfc.score(cv_test_features, test_label_names)
print('Test Accuracy:', rfc_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.52052011  0.51669381  0.53485528  0.51327072
0.5212766 ]
Mean CV Accuracy: 0.521323304518
Test Accuracy: 0.52987915908
```

```
# Gradient Boosting Machines
from sklearn.ensemble import GradientBoostingClassifier

gbc = GradientBoostingClassifier(n_estimators=10, random_state=42)
gbc.fit(cv_train_features, train_label_names)
gbc_bow_cv_scores = cross_val_score(gbc, cv_train_features, train_label_
names, cv=5)
gbc_bow_cv_mean_score = np.mean(gbc_bow_cv_scores)
print('CV Accuracy (5-fold):', gbc_bow_cv_scores)
print('Mean CV Accuracy:', gbc_bow_cv_mean_score)
gbc_bow_test_score = gbc.score(cv_test_features, test_label_names)
print('Test Accuracy:', gbc_bow_test_score)
```

```
CV Accuracy (5-fold): [ 0.55424624  0.53827362  0.54219323  0.55206207
0.55441899]
Mean CV Accuracy: 0.548238828239
Test Accuracy: 0.547922529383
```

It is interesting to see that simpler models like Naïve Bayes and Logistic Regression performed much better than the ensemble models. Let's look at the next model pipeline now.

TF-IDF Features with Classification Models

We use TF-IDF features to train our classification models. Assuming TF-IDF weighs down unimportant features, we might get better performing models. Let's test our assumption!

```
from sklearn.feature_extraction.text import TfidfVectorizer

# build BOW features on train articles
tv = TfidfVectorizer(use_idf=True, min_df=0.0, max_df=1.0)
tv_train_features = tv.fit_transform(train_corpus)

# transform test articles into features
tv_test_features = tv.transform(test_corpus)

print('TFIDF model:> Train features shape:', tv_train_features.shape,
      ' Test features shape:', tv_test_features.shape)
TFIDF model:> Train features shape: (12263, 66865)  Test features shape:
(6041, 66865)
```

We now build several classifiers on these features using the training data and test their performance on the test dataset using all the classification models. We also check model accuracies using five-fold cross validation, just like we did earlier.

```
# Naïve Bayes
mnb = MultinomialNB(alpha=1)
mnb.fit(tv_train_features, train_label_names)
mnb_tfidf_cv_scores = cross_val_score(mnb, tv_train_features, train_label_
names, cv=5)
mnb_tfidf_cv_mean_score = np.mean(mnb_tfidf_cv_scores)
```

```
print('CV Accuracy (5-fold):', mnb_tfidf_cv_scores)
print('Mean CV Accuracy:', mnb_tfidf_cv_mean_score)
mnb_tfidf_test_score = mnb.score(tv_test_features, test_label_names)
print('Test Accuracy:', mnb_tfidf_test_score)
```

```
CV Accuracy (5-fold): [ 0.71759447  0.70969055  0.71585813  0.7121274
0.7111293 ]
Mean CV Accuracy: 0.713279971122
Test Accuracy: 0.713954643271
```

```
# Logistic Regression
lr = LogisticRegression(penalty='l2', max_iter=100, C=1, random_state=42)
lr.fit(tv_train_features, train_label_names)
lr_tfidf_cv_scores = cross_val_score(lr, tv_train_features, train_label_
names, cv=5)
lr_tfidf_cv_mean_score = np.mean(lr_tfidf_cv_scores)
print('CV Accuracy (5-fold):', lr_tfidf_cv_scores)
print('Mean CV Accuracy:', lr_tfidf_cv_mean_score)
lr_tfidf_test_score = lr.score(tv_test_features, test_label_names)
print('Test Accuracy:', lr_tfidf_test_score)
```

```
CV Accuracy (5-fold): [ 0.74725721  0.73493485  0.73257236  0.74520212
0.73076923]
Mean CV Accuracy: 0.738147156079
Test Accuracy: 0.745240854163
```

```
# Support Vector Machines
svm = LinearSVC(penalty='l2', C=1, random_state=42)
svm.fit(tv_train_features, train_label_names)
svm_tfidf_cv_scores = cross_val_score(svm, tv_train_features, train_label_
names, cv=5)
svm_tfidf_cv_mean_score = np.mean(svm_tfidf_cv_scores)
print('CV Accuracy (5-fold):', svm_tfidf_cv_scores)
print('Mean CV Accuracy:', svm_tfidf_cv_mean_score)
svm_tfidf_test_score = svm.score(tv_test_features, test_label_names)
print('Test Accuracy:', svm_tfidf_test_score)
```

```
CV Accuracy (5-fold): [ 0.76635514  0.7536645
0.75743987  0.76439363  0.75695581]
Mean CV Accuracy: 0.75976178901
Test Accuracy: 0.762456546929
```

```
# SVM with Stochastic Gradient Descent
svm_sgd = SGDClassifier(loss='hinge', penalty='l2', max_iter=5, random_
state=42)
svm_sgd.fit(tv_train_features, train_label_names)
svmsgd_tfidf_cv_scores = cross_val_score(svm_sgd, tv_train_features, train_
label_names, cv=5)
svmsgd_tfidf_cv_mean_score = np.mean(svmsgd_tfidf_cv_scores)
print('CV Accuracy (5-fold):', svmsgd_tfidf_cv_scores)
print('Mean CV Accuracy:', svmsgd_tfidf_cv_mean_score)
svmsgd_tfidf_test_score = svm_sgd.score(tv_test_features, test_label_names)
print('Test Accuracy:', svmsgd_tfidf_test_score)
```

```
CV Accuracy (5-fold): [ 0.76513612  0.75570033  0.75377089  0.76112699
0.75695581]
Mean CV Accuracy: 0.75853802856
Test Accuracy: 0.765767257077
```

```
# Random Forest
rfc = RandomForestClassifier(n_estimators=10, random_state=42)
rfc.fit(tv_train_features, train_label_names)
rfc_tfidf_cv_scores = cross_val_score(rfc, tv_train_features, train_label_
names, cv=5)
rfc_tfidf_cv_mean_score = np.mean(rfc_tfidf_cv_scores)
print('CV Accuracy (5-fold):', rfc_tfidf_cv_scores)
print('Mean CV Accuracy:', rfc_tfidf_cv_mean_score)
rfc_tfidf_test_score = rfc.score(tv_test_features, test_label_names)
print('Test Accuracy:', rfc_tfidf_test_score)
```

```
CV Accuracy (5-fold): [ 0.53596099  0.5252443
0.53852426  0.51204573  0.54296236]
Mean CV Accuracy: 0.53094752738
Test Accuracy: 0.545936103294
```

```
# Gradient Boosting
gbc = GradientBoostingClassifier(n_estimators=10, random_state=42)
gbc.fit(tv_train_features, train_label_names)
gbc_tfidf_cv_scores = cross_val_score(gbc, tv_train_features, train_label_
names, cv=5)
gbc_tfidf_cv_mean_score = np.mean(gbc_tfidf_cv_scores)
print('CV Accuracy (5-fold):', gbc_tfidf_cv_scores)
print('Mean CV Accuracy:', gbc_tfidf_cv_mean_score)
gbc_tfidf_test_score = gbc.score(tv_test_features, test_label_names)
print('Test Accuracy:', gbc_tfidf_test_score)
```

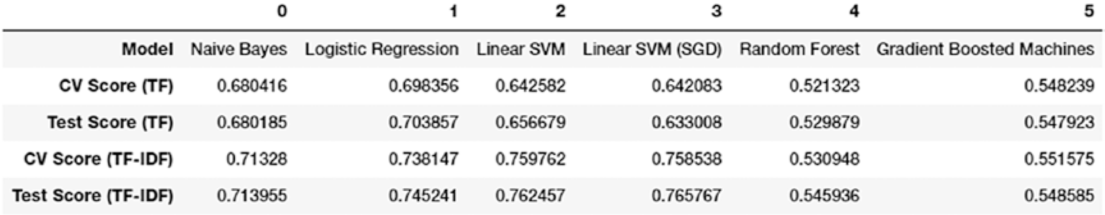
```
CV Accuracy (5-fold): [ 0.55790329  0.53827362  0.55768447  0.55859535  0.54541735]
Mean CV Accuracy: 0.551574813725
Test Accuracy: 0.548584671412
```

It's interesting to see that the overall accuracy of several models increases by quite a bit, including logistic regression, Naïve Bayes, and SVM. Interestingly, the ensemble models don't perform as well. Using more estimators might improve them, but still wouldn't be as good as the other models and it would take a huge amount of training time.

Comparative Model Performance Evaluation

We can now do a nice comparison of all the models we have tried so far with the two different feature engineering techniques. We will build a dataframe from our modeling results and compare the results. See Figure 5-10.

```
pd.DataFrame([['Naive Bayes', mnb_bow_cv_mean_score, mnb_bow_test_score,
               mnb_tfidf_cv_mean_score, mnb_tfidf_test_score],
              ['Logistic Regression', lr_bow_cv_mean_score, lr_bow_test_
              score, lr_tfidf_cv_mean_score, lr_tfidf_test_score],
              ['Linear SVM', svm_bow_cv_mean_score, svm_bow_test_score,
               svm_tfidf_cv_mean_score, svm_tfidf_test_score],
              ['Linear SVM (SGD)', svmsgd_bow_cv_mean_score, svmsgd_bow_test_
              score, svmsgd_tfidf_cv_mean_score, svmsgd_tfidf_test_score],
              ['Random Forest', rfc_bow_cv_mean_score, rfc_bow_test_score,
               rfc_tfidf_cv_mean_score, rfc_tfidf_test_score],
              ['Gradient Boosted Machines', gbc_bow_cv_mean_score, gbc_bow_
              test_score, gbc_tfidf_cv_mean_score, gbc_tfidf_test_score]],
              columns=['Model', 'CV Score (TF)', 'Test Score (TF)',
                      'CV Score (TF-IDF)', 'Test Score (TF-IDF)'],
             ).T
```



	0	1	2	3	4	5
Model	Naive Bayes	Logistic Regression	Linear SVM	Linear SVM (SGD)	Random Forest	Gradient Boosted Machines
CV Score (TF)	0.680416	0.698356	0.642582	0.642083	0.521323	0.548239
Test Score (TF)	0.680185	0.703857	0.656679	0.633008	0.529879	0.547923
CV Score (TF-IDF)	0.71328	0.738147	0.759762	0.758538	0.530948	0.551575
Test Score (TF-IDF)	0.713955	0.745241	0.762457	0.765767	0.545936	0.548585

Figure 5-10. *Comparative model performance evaluation*

Figure 5-10 clearly shows us that the best performing models were SVM followed by Logistic Regression and Naïve Bayes. Ensemble models didn't perform as well on this dataset.

Word2Vec Embeddings with Classification Models

Let's try using the newer advanced feature engineering techniques with our classification models. We start by generating Word2Vec embeddings. An important point to note here is that word embedding models generate a dense embedding vector of fixed lengths for each word. Hence, we need some scheme to generate fixed embeddings for each document. One way is to average the word embeddings for all the words in the document (or even take the TF-IDF weighted average!). Let's build a scheme to generate document embeddings from the averaged word embeddings.

```
def document_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)

    def average_word_vectors(words, model, vocabulary, num_features):
        feature_vector = np.zeros((num_features,), dtype="float64")
        nwords = 0.
        for word in words:
            if word in vocabulary:
                nwords = nwords + 1.
                feature_vector = np.add(feature_vector, model.wv[word])
        if nwords:
            feature_vector = np.divide(feature_vector, nwords)

        return feature_vector

    features = [average_word_vectors(tokenized_sentence, model, vocabulary,
                num_features) for tokenized_sentence in corpus]
    return np.array(features)
```

We use Gensim, an excellent Python framework, to generate Word2Vec embeddings for all words in our corpus.

```
# tokenize corpus
tokenized_train = [tn.tokenizer.tokenize(text)
                   for text in train_corpus]
tokenized_test = [tn.tokenizer.tokenize(text)
                   for text in test_corpus]

# generate word2vec word embeddings
import gensim
# build word2vec model
w2v_num_features = 1000
w2v_model = gensim.models.Word2Vec(tokenized_train, size=w2v_num_features,
window=100, min_count=2, sample=1e-3, sg=1, iter=5, workers=10)

# generate document level embeddings
# remember we only use train dataset vocabulary embeddings
# so that test dataset truly remains an unseen dataset
# generate averaged word vector features from word2vec model
avg_wv_train_features = document_vectorizer(corpus=tokenized_train,
model=w2v_model, num_features=w2v_num_features)
```

```
avg_wv_test_features = document_vectorizer(corpus=tokenized_test,
model=w2v_model, num_features=w2v_num_features)

print('Word2Vec model:> Train features shape:', avg_wv_train_features.
shape,' Test features shape:', avg_wv_test_features.shape)

Word2Vec model:> Train features shape: (12263, 1000)  Test features shape:
(6041, 1000)
```

Let's try using one of our best models, SVM with SGD, to check the model performance on the test data.

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import SGDClassifier

svm = SGDClassifier(loss='hinge', penalty='l2', random_state=42, max_
iter=500)
svm.fit(avg_wv_train_features, train_label_names)
svm_w2v_cv_scores = cross_val_score(svm, avg_wv_train_features, train_
label_names, cv=5)
svm_w2v_cv_mean_score = np.mean(svm_w2v_cv_scores)
print('CV Accuracy (5-fold):', svm_w2v_cv_scores)
print('Mean CV Accuracy:', svm_w2v_cv_mean_score)
svm_w2v_test_score = svm.score(avg_wv_test_features, test_label_names)
print('Test Accuracy:', svm_w2v_test_score)

CV Accuracy (5-fold): [0.76026006 0.74796417 0.73746433 0.73989383 0.74386252]
Mean CV Accuracy: 0.7458889820674891
Test Accuracy: 0.7381228273464658
```

Definitely a good model performance but not better than our TF-IDF based model, which gave much better test accuracy.

GloVe Embeddings with Classification Models

We now generate GloVe-based word embeddings for each word, generate document-level embeddings, and use our SVM model to test the model performance. We use spaCy's default word embeddings generated from the common crawl corpus.

```
# feature engineering with GloVe model
train_nlp = [tn.nlp(item) for item in train_corpus]
train_glove_features = np.array([item.vector for item in train_nlp])

test_nlp = [tn.nlp(item) for item in test_corpus]
test_glove_features = np.array([item.vector for item in test_nlp])

print('GloVe model:> Train features shape:', train_glove_features.shape,
      ' Test features shape:', test_glove_features.shape)

GloVe model:> Train features shape: (12263, 300)  Test features shape:
(6041, 300)

# Building our SVM model
svm = SGDClassifier(loss='hinge', penalty='l2', random_state=42, max_
iter=500)
svm.fit(train_glove_features, train_label_names)
svm_glove_cv_scores = cross_val_score(svm, train_glove_features, train_
label_names, cv=5)
svm_glove_cv_mean_score = np.mean(svm_glove_cv_scores)
print('CV Accuracy (5-fold):', svm_glove_cv_scores)
print('Mean CV Accuracy:', svm_glove_cv_mean_score)
svm_glove_test_score = svm.score(test_glove_features, test_label_names)
print('Test Accuracy:', svm_glove_test_score)

CV Accuracy (5-fold): [ 0.68996343  0.67711726  0.67101508  0.67006942
0.66448445]
Mean CV Accuracy: 0.674529928944
Test Accuracy: 0.666777023672
```

It looks like the performance is not as good and that could be because we're using pre-generated word embeddings. Let's now take a look at FastText!

FastText Embeddings with Classification Models

We now leverage Gensim again, but use Facebook's FastText model to generate word embeddings from which we will build our document embeddings.

```
from gensim.models.fasttext import FastText

ft_num_features = 1000
# sg decides whether to use the skip-gram model (1) or CBOW (0)
ft_model = FastText(tokenized_train, size=ft_num_features, window=100,
                    min_count=2, sample=1e-3, sg=1, iter=5, workers=10)

# generate averaged word vector features from word2vec model
avg_ft_train_features = document_vectorizer(corpus=tokenized_train,
model=ft_model, num_features=ft_num_features)
avg_ft_test_features = document_vectorizer(corpus=tokenized_test,
model=ft_model, num_features=ft_num_features)

print('FastText model:> Train features shape:', avg_ft_train_features.shape,
      ' Test features shape:', avg_ft_test_features.shape)

FastText model:> Train features shape: (12263, 1000)  Test features shape:
(6041, 1000)
```

Now, just like the previous pipelines, we train and evaluate our SVM model on these features.

```
svm = SGDClassifier(loss='hinge', penalty='l2', random_state=42, max_iter=500)
svm.fit(avg_ft_train_features, train_label_names)
svm_ft_cv_scores = cross_val_score(svm, avg_ft_train_features, train_label_
names, cv=5)
svm_ft_cv_mean_score = np.mean(svm_ft_cv_scores)
print('CV Accuracy (5-fold):', svm_ft_cv_scores)
print('Mean CV Accuracy:', svm_ft_cv_mean_score)
svm_ft_test_score = svm.score(avg_ft_test_features, test_label_names)
print('Test Accuracy:', svm_ft_test_score)
```

```
CV Accuracy (5-fold): [0.76391711 0.74307818 0.74194863 0.74724377
0.74795417]
Mean CV Accuracy: 0.7488283727085712
Test Accuracy: 0.7434199635821884
```

This is definitely the best performing model out of all the word embedding based models, but it's still not better than our TF-IDF based model. Let's quickly build a two-hidden layer neural network and see if we get a better model performance.

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(solver='adam', alpha=1e-5, learning_rate='adaptive',
early_stopping=True, activation = 'relu', hidden_layer_sizes=(512, 512),
random_state=42)
mlp.fit(avg_ft_train_features, train_label_names)

svm_ft_test_score = mlp.score(avg_ft_test_features, test_label_names)
print('Test Accuracy:', svm_ft_test_score)
```

```
Test Accuracy: 0.7328256911107432
```

What does this tell us? Word embedding models or deep learning models might be good, but that doesn't mean they are a silver bullet for all our problems. Often traditional models might out-perform them, depending on the problem and the context!

Model Tuning

Model tuning is perhaps one of the key stages in the machine learning process and can lead to better performing models. Any machine learning model typically has hyperparameters, which are high-level concepts much like configuration settings that you can tune like knobs in a device! A very important point to remember is that hyperparameters are model parameters that are not directly learned within estimators and do not depend on the underlying data (as opposed to model parameters or coefficients like the coefficients of logistic regression, which can change based on the underlying training data).

It is possible and recommended to search the hyperparameter space for the best cross-validation score for which we use a five-fold cross validation scheme along with grid search for finding the best hyperparameter values. A typical search for the best hyperparameter values during tuning consists of the following major components:

- A model or estimator like LogisticRegression from Scikit-Learn
- A hyperparameter space that we can define with values and ranges
- A method for searching or sampling candidates like Grid Search
- A cross-validation scheme, like five-fold cross-validation
- A score function, like accuracy, for classification models

There are two very common approaches for sampling search candidates also available in Scikit-Learn. We have `GridSearchCV`, which exhaustively considers all parameter combinations set by users. However, `RandomizedSearchCV` typically samples a given number of candidates from a parameter space with a specified distribution instead of taking all combinations. We use Grid Search for our tuning experiments.

To tune the experiments, we also use a Scikit-Learn `Pipeline` object, which is an excellent way to chain multiple components together where we sequentially apply a list of transforms like data preprocessors, feature engineering methods, and a model estimator for predictions. Intermediate steps of the pipeline must be some form of a "transformer," that is, they must implement fit and transform methods.

The purpose of the pipeline and why we want to use it is so that we can assemble multiple components like feature engineering and modeling so that they can be cross-validated while setting different hyperparameter values for grid search. Let's get started with tuning our Naïve Bayes model!

```
# Tuning our Multinomial Naïve Bayes model
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfVectorizer

mnb_pipeline = Pipeline([('tfidf', TfidfVectorizer()),
                         ('mnb', MultinomialNB())
                        ])
```

```
param_grid = {'tfidf__ngram_range': [(1, 1), (1, 2)],
              'mnb__alpha': [1e-5, 1e-4, 1e-2, 1e-1, 1]
}

gs_mnb = GridSearchCV(mnb_pipeline, param_grid, cv=5, verbose=2)
gs_mnb = gs_mnb.fit(train_corpus, train_label_names)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
[CV] mnb__alpha=1e-05, tfidf__ngram_range=(1, 1) .....................
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[CV] ...... mnb__alpha=1e-05, tfidf__ngram_range=(1, 1), total=   1.5s
[CV] mnb__alpha=1e-05, tfidf__ngram_range=(1, 1) .....................
...
...
[CV] mnb__alpha=1, tfidf__ngram_range=(1, 2) .........................
[CV] .......... mnb__alpha=1, tfidf__ngram_range=(1, 2), total=   6.2s
[Parallel(n_jobs=1)]: Done  50 out of  50 | elapsed:  4.7min finished
```

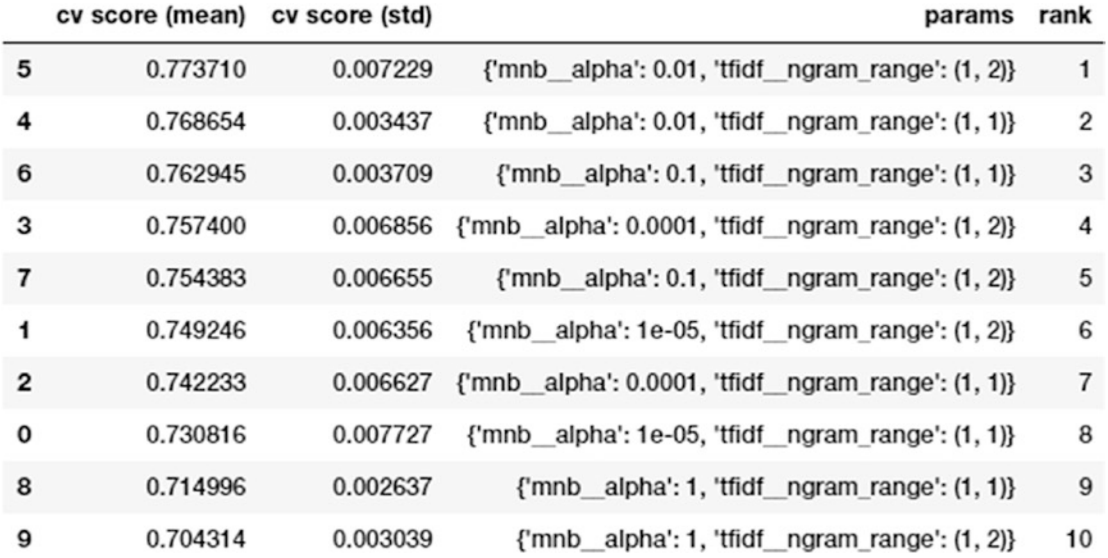
We can now inspect the hyperparameter values chosen for our best estimator/model using the following code.

```
gs_mnb.best_estimator_.get_params()
```

```
{'memory': None,
 'steps': [('tfidf',
   TfidfVectorizer(analyzer='word', max_df=1.0, min_df=1, ngram_range=(1, 2),
                   norm='l2', ...,  use_idf=True),
  ('mnb', MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True))],
 'tfidf': TfidfVectorizer(analyzer='word', max_df=1.0, min_df=1, ngram_
 range=(1, 2),
                          norm='l2', ...,  use_idf=True),
 'mnb': MultinomialNB(alpha=0.01, class_prior=None, fit_prior=True),
 'tfidf__analyzer': 'word', 'tfidf__binary': False, 'tfidf__decode_error':
 'strict',
 'tfidf__dtype': numpy.float64, 'tfidf__encoding': 'utf-8', 'tfidf__input':
 'content',
 'tfidf__lowercase': True, 'tfidf__max_df': 1.0, 'tfidf__max_features': None,
 'tfidf__min_df': 1, 'tfidf__ngram_range': (1, 2), 'tfidf__norm': 'l2',
 'tfidf__preprocessor': None, 'tfidf__smooth_idf': True, 'tfidf__stop_
 words': None,
 'tfidf__strip_accents': None, 'tfidf__sublinear_tf': False,
 'tfidf__token_pattern': '(?u)\\b\\w\\w+\\b', 'tfidf__tokenizer': None,
 'tfidf__use_idf': True,
 'tfidf__vocabulary': None, 'mnb__alpha': 0.01, 'mnb__class_prior': None,
 'mnb__fit_prior': True}
```

Now you might be wondering how these hyperparameters specifically were selected for the best estimator. Well, it decided this based on the model performance, with those hyperparameter values on the five-folds of validation data during cross-validation. See Figure 5-11.

```
cv_results = gs_mnb.cv_results_
results_df = pd.DataFrame({'rank': cv_results['rank_test_score'],
                           'params': cv_results['params'],
                           'cv score (mean)': cv_results['mean_test_score'],
                           'cv score (std)': cv_results['std_test_score']}
              )
results_df = results_df.sort_values(by=['rank'], ascending=True)
pd.set_option('display.max_colwidth', 100)
results_df
```



	cv score (mean)	cv score (std)	params	rank
5	0.773710	0.007229	{'mnb__alpha': 0.01, 'tfidf__ngram_range': (1, 2)}	1
4	0.768654	0.003437	{'mnb__alpha': 0.01, 'tfidf__ngram_range': (1, 1)}	2
6	0.762945	0.003709	{'mnb__alpha': 0.1, 'tfidf__ngram_range': (1, 1)}	3
3	0.757400	0.006856	{'mnb__alpha': 0.0001, 'tfidf__ngram_range': (1, 2)}	4
7	0.754383	0.006655	{'mnb__alpha': 0.1, 'tfidf__ngram_range': (1, 2)}	5
1	0.749246	0.006356	{'mnb__alpha': 1e-05, 'tfidf__ngram_range': (1, 2)}	6
2	0.742233	0.006627	{'mnb__alpha': 0.0001, 'tfidf__ngram_range': (1, 1)}	7
0	0.730816	0.007727	{'mnb__alpha': 1e-05, 'tfidf__ngram_range': (1, 1)}	8
8	0.714996	0.002637	{'mnb__alpha': 1, 'tfidf__ngram_range': (1, 1)}	9
9	0.704314	0.003039	{'mnb__alpha': 1, 'tfidf__ngram_range': (1, 2)}	10

Figure 5-11. *Model performances across different hyperparameter values in the hyperparameter space*

From the table in Figure 5-11, you can see how the best hyperparameters including bi-gram TF-IDF features gave the best cross-validation accuracy. Note that we are never tuning our models based on test data scores, because that would end up biasing our model toward the test dataset. We can now check our tuned model's performance on the test data.

```
best_mnb_test_score = gs_mnb.score(test_corpus, test_label_names)
print('Test Accuracy :', best_mnb_test_score)
```

```
Test Accuracy : 0.7735474259228604
```

Looks like we have achieved a model accuracy of 77.3%, which is an improvement of 6% over the base model! Let's look at how it performs for logistic regression now.

```
# Tuning our Logistic Regression model
lr_pipeline = Pipeline([('tfidf', TfidfVectorizer()),
                        ('lr', LogisticRegression(penalty='l2', max_
                        iter=100, random_state=42))
                       ])
```

```
param_grid = {'tfidf__ngram_range': [(1, 1), (1, 2)],
              'lr__C': [1, 5, 10]
}

gs_lr = GridSearchCV(lr_pipeline, param_grid, cv=5, verbose=2)
gs_lr = gs_lr.fit(train_corpus, train_label_names)

Fitting 5 folds for each of 6 candidates, totalling 30 fits
[CV] lr__C=1, tfidf__ngram_range=(1, 1) ..............................
[CV] ............... lr__C=1, tfidf__ngram_range=(1, 1), total=   5.9s
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    7.1s remaining:    0.0s
[CV] lr__C=1, tfidf__ngram_range=(1, 1) ..............................
[CV] ............... lr__C=1, tfidf__ngram_range=(1, 1), total=   6.5s
...
...
[CV] lr__C=10, tfidf__ngram_range=(1, 2) .............................
[CV] .............. lr__C=10, tfidf__ngram_range=(1, 2), total=  47.8s
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed: 13.0min finished

# evaluate best tuned model on the test dataset
best_lr_test_score = gs_lr.score(test_corpus, test_label_names)
print('Test Accuracy :', best_lr_test_score)

Test Accuracy : 0.766926005628
```

We get an overall test accuracy of approximately 77%, which is almost a 2.5% improvement from the base logistic regression model. Finally, let's tune our top two SVM models—the regular Linear SVM model and the SVM with Stochastic Gradient Descent.

```
# Tuning the Linear SVM model
svm_pipeline = Pipeline([('tfidf', TfidfVectorizer()),
                         ('svm', LinearSVC(random_state=42))
                        ])

param_grid = {'tfidf__ngram_range': [(1, 1), (1, 2)],
              'svm__C': [0.01, 0.1, 1, 5]
}
```

```
gs_svm = GridSearchCV(svm_pipeline, param_grid, cv=5, verbose=2)
gs_svm = gs_svm.fit(train_corpus, train_label_names)

# evaluating best tuned model on the test dataset
best_svm_test_score = gs_svm.score(test_corpus, test_label_names)
print('Test Accuracy :', best_svm_test_score)
```

```
Test Accuracy : 0.77685813607
```

This is definitely the highest overall accuracy we have obtained so far! However, not a huge improvement from the default linear SVM model performance. The SVM with SGD gives us a tuned model accuracy of 76.8%.

Model Performance Evaluation

Choosing the best model for deployment depends on a number of factors, like the model speed, accuracy, ease of use, understanding, and so on. Based on all the models we have built, the Naïve Bayes model is the fastest to train and, even though the SVM model might be slightly better on the test dataset in terms of accuracy, SVMs are notoriously slow and often hard to scale. Let's take a detailed performance evaluation of our best, tuned Naïve Bayes model on the test dataset. We use our nifty `model_evaluation_utils` module for the purpose of model evaluation.

```
import model_evaluation_utils as meu

mnb_predictions = gs_mnb.predict(test_corpus)
unique_classes = list(set(test_label_names))
meu.get_metrics(true_labels=test_label_names, predicted_labels=mnb_
predictions)
```

```
Accuracy: 0.7735
Precision: 0.7825
Recall: 0.7735
F1 Score: 0.7696
```

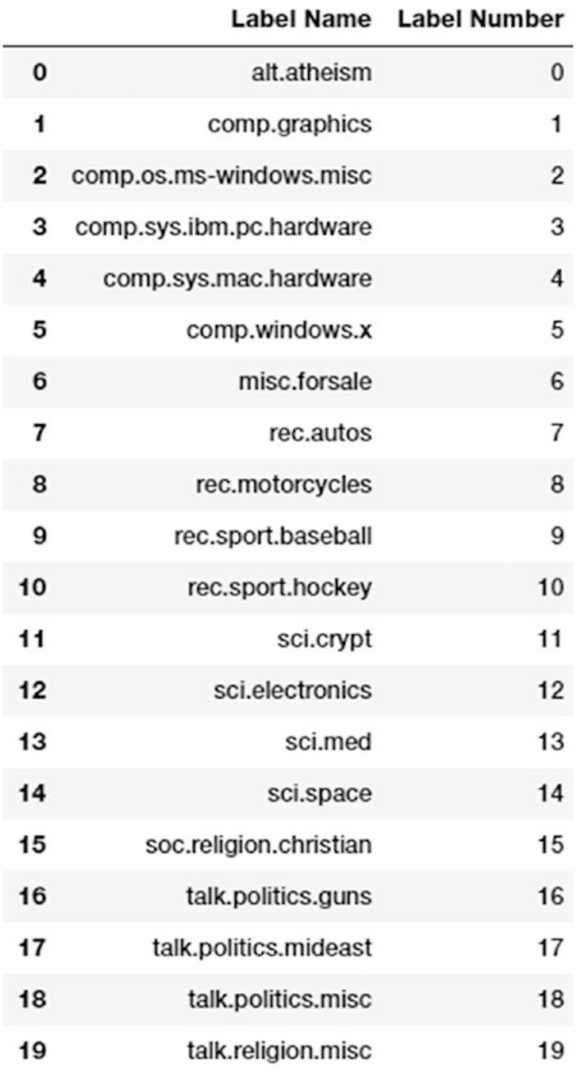
It is good to see good consistency with the classification metrics. Besides seeing the holistic view of model performance metrics, often a more granular view into per-class model performance metrics helps. Let's take a look at that.

```
meu.display_classification_report(true_labels=test_label_names,
                                  predicted_labels=mnb_predictions,
                                  classes=unique_classes)
```

```
                          precision    recall  f1-score   support

 comp.os.ms-windows.misc       0.76      0.72      0.74       315
      talk.politics.misc       0.72      0.68      0.70       244
           comp.graphics       0.64      0.75      0.69       289
          comp.windows.x       0.79      0.84      0.81       287
      talk.religion.misc       0.67      0.21      0.32       199
comp.sys.ibm.pc.hardware       0.69      0.76      0.72       324
   comp.sys.mac.hardware       0.78      0.77      0.77       295
               sci.crypt       0.79      0.85      0.82       302
   talk.politics.mideast       0.85      0.87      0.86       326
            misc.forsale       0.83      0.77      0.80       314
                 sci.med       0.88      0.88      0.88       322
         rec.motorcycles       0.88      0.74      0.80       351
         sci.electronics       0.80      0.72      0.76       307
        rec.sport.hockey       0.88      0.92      0.90       308
      talk.politics.guns       0.65      0.81      0.72       281
               sci.space       0.84      0.81      0.83       324
      rec.sport.baseball       0.94      0.88      0.91       336
             alt.atheism       0.80      0.57      0.67       268
               rec.autos       0.82      0.74      0.78       328
  soc.religion.christian       0.57      0.92      0.70       321

               micro avg       0.77      0.77      0.77      6041
               macro avg       0.78      0.76      0.76      6041
            weighted avg       0.78      0.77      0.77      6041
```

This gives us a nice overview into the model performance for each newsgroup class and interestingly some categories like religion, Christianity, and atheism have slightly lower performance. Could it be that the model is getting some of these mixed up? The confusion matrix is a great way to test this assumption. Let's first look at the newsgroup name to number mappings. See Figure 5-12.

```
label_data_map = {v:k for k, v in data_labels_map.items()}
label_map_df = pd.DataFrame(list(label_data_map.items()),
                            columns=['Label Name', 'Label Number'])
label_map_df
```

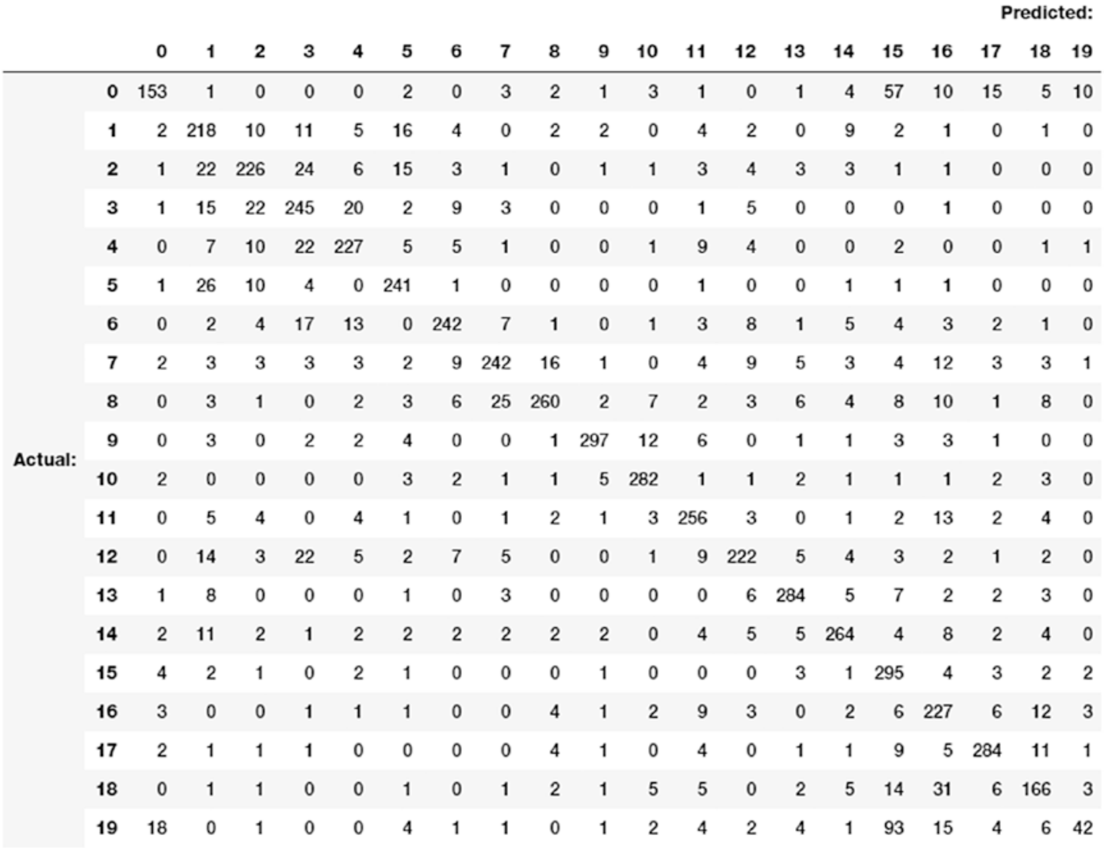


	Label Name	Label Number
0	alt.atheism	0
1	comp.graphics	1
2	comp.os.ms-windows.misc	2
3	comp.sys.ibm.pc.hardware	3
4	comp.sys.mac.hardware	4
5	comp.windows.x	5
6	misc.forsale	6
7	rec.autos	7
8	rec.motorcycles	8
9	rec.sport.baseball	9
10	rec.sport.hockey	10
11	sci.crypt	11
12	sci.electronics	12
13	sci.med	13
14	sci.space	14
15	soc.religion.christian	15
16	talk.politics.guns	16
17	talk.politics.mideast	17
18	talk.politics.misc	18
19	talk.religion.misc	19

Figure 5-12. *Mapping between class label names and numbers*

We can now build a confusion matrix to show the correct and misclassified instances of each class label, which we represent by numbers for display purposes, due to the long names. See Figure 5-13.

```
unique_class_nums = label_map_df['Label Number'].values
mnb_prediction_class_nums = [label_data_map[item] for item in mnb_predictions]
meu.display_confusion_matrix_pretty(true_labels=test_label_nums,
                                   predicted_labels=mnb_prediction_class_
                                   nums, classes=unique_class_nums)
```

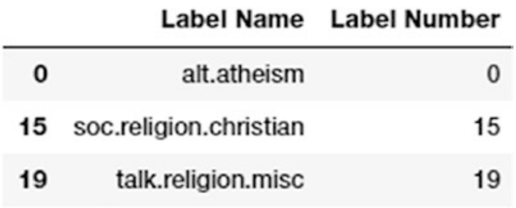


		Predicted:																			
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Actual:	0	153	1	0	0	0	2	0	3	2	1	3	1	0	1	4	57	10	15	5	10
	1	2	218	10	11	5	16	4	0	2	2	0	4	2	0	9	2	1	0	1	0
	2	1	22	226	24	6	15	3	1	0	1	1	3	4	3	3	1	1	0	0	0
	3	1	15	22	245	20	2	9	3	0	0	0	1	5	0	0	0	1	0	0	0
	4	0	7	10	22	227	5	5	1	0	0	1	9	4	0	0	2	0	0	1	1
	5	1	26	10	4	0	241	1	0	0	0	0	1	0	0	1	1	1	0	0	0
	6	0	2	4	17	13	0	242	7	1	0	1	3	8	1	5	4	3	2	1	0
	7	2	3	3	3	3	2	9	242	16	1	0	4	9	5	3	4	12	3	3	1
	8	0	3	1	0	2	3	6	25	260	2	7	2	3	6	4	8	10	1	8	0
	9	0	3	0	2	2	4	0	0	1	297	12	6	0	1	1	3	3	1	0	0
	10	2	0	0	0	0	3	2	1	1	5	282	1	1	2	1	1	1	2	3	0
	11	0	5	4	0	4	1	0	1	2	1	3	256	3	0	1	2	13	2	4	0
	12	0	14	3	22	5	2	7	5	0	0	1	9	222	5	4	3	2	1	2	0
	13	1	8	0	0	0	1	0	3	0	0	0	0	6	284	5	7	2	2	3	0
	14	2	11	2	1	2	2	2	2	2	2	0	4	5	5	264	4	8	2	4	0
	15	4	2	1	0	2	1	0	0	0	1	0	0	0	3	1	295	4	3	2	2
	16	3	0	0	1	1	1	0	0	4	1	2	9	3	0	2	6	227	6	12	3
	17	2	1	1	1	0	0	0	0	4	1	0	4	0	1	1	9	5	284	11	1
	18	0	1	1	0	0	1	0	1	2	1	5	5	0	2	5	14	31	6	166	3
	19	18	0	1	0	0	4	1	1	0	1	2	4	2	4	1	93	15	4	6	42

***Figure 5-13.** Confusion matrix for the predictions of our Naïve Bayes model on test data*

The diagonal of our confusion matrix has the meat of the numbers, which indicates that most of our predictions match the actual class labels! Interestingly, class labels 0, 15, and 19 seem to have a lot of misclassifications. Let's take a closer look at these class labels to see what their newsgroup names are. See Figure 5-14.

```
label_map_df[label_map_df['Label Number'].isin([0, 15, 19])]
```



	Label Name	Label Number
0	alt.atheism	0
15	soc.religion.christian	15
19	talk.religion.misc	19

Figure 5-14. *Class label numbers and names for misclassified newsgroups*

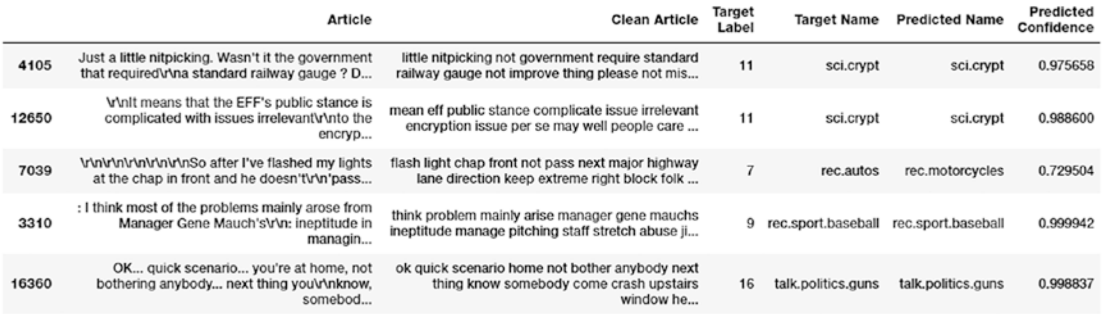
Just like we suspected, all the newsgroups pertaining to different aspects of religion have more misclassifications, which that indicates the model must be misclassifying instances of one of these classes. Let's dive a bit deeper into this and explore some specific instances.

```
# Extract test document row numbers
train_idx, test_idx = train_test_split(np.array(range(len(data_df
['Article']))), test_size=0.33, random_state=42)
test_idx
```

```
array([ 4105, 12650,  7039, ...,  4772,  7803,  9616])
```

We now add two columns to our dataframe in our test dataset. The first column is the predicted label from our Naïve Bayes model and the second column is the confidence of the model when making the prediction, which is basically the probability of the model prediction. See Figure 5-15.

```
predict_probas = gs_mnb.predict_proba(test_corpus).max(axis=1)
test_df = data_df.iloc[test_idx]
test_df['Predicted Name'] = mnb_predictions
test_df['Predicted Confidence'] = predict_probas
test_df.head()
```

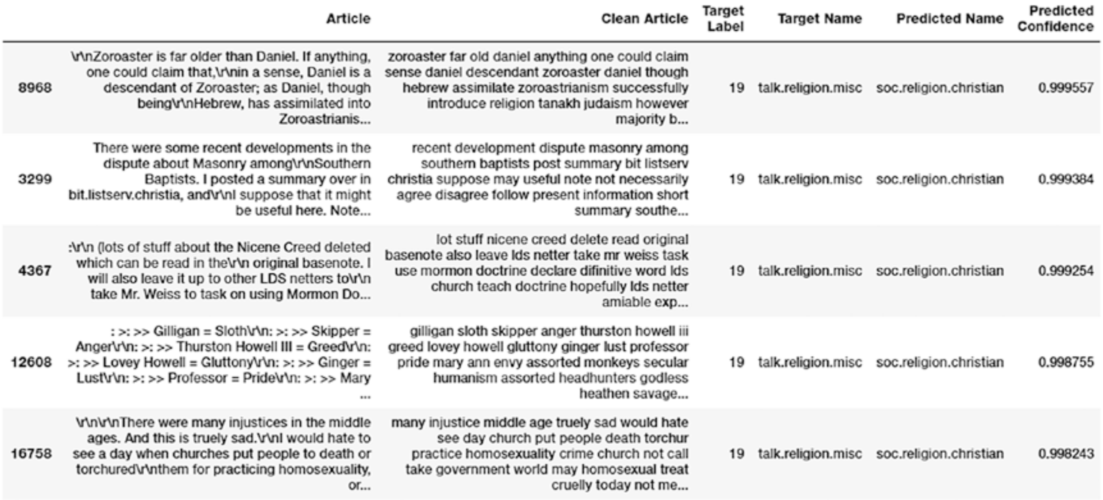


	Article	Clean Article	Target Label	Target Name	Predicted Name	Predicted Confidence
4105	Just a little nitpicking. Wasn't it the government that required\r\na standard railway gauge ? D...	little nitpicking not government require standard railway gauge not improve thing please not mis...	11	sci.crypt	sci.crypt	0.975658
12650	\r\nIt means that the EFF's public stance is complicated with issues irrelevant\r\nto the encryp...	mean eff public stance complicate issue irrelevant encryption issue per se may well people care ...	11	sci.crypt	sci.crypt	0.988600
7039	\r\n\r\n\r\n\r\n\r\nSo after I've flashed my lights at the chap in front and he doesn't\r\n'pass...	flash light chap front not pass next major highway lane direction keep extreme right block folk ...	7	rec.autos	rec.motorcycles	0.729504
3310	: I think most of the problems mainly arose from Manager Gene Mauch's\r\n: ineptitude in managin...	think problem mainly arise manager gene mauchs ineptitude manage pitching staff stretch abuse ji...	9	rec.sport.baseball	rec.sport.baseball	0.999942
16360	OK... quick scenario... you're at home, not bothering anybody... next thing you\r\nknow, somebod...	ok quick scenario home not bother anybody next thing know somebody come crash upstairs window he...	16	talk.politics.guns	talk.politics.guns	0.998837

Figure 5-15. *Adding additional metadata to our test dataset with model predictions and confidence scores*

Based on the dataframe snapshot depicted in Figure 5-15, it looks like everything is in order with the test dataset articles, actual labels, predicted labels, and confidence scores. Let's now take a look at some articles that were from the newsgroup `talk.religion.misc`, but our model predicted `soc.religion.christian` with the highest confidence. See Figure 5-16.

```
pd.set_option('display.max_colwidth', 200)
res_df = (test_df[(test_df['Target Name'] == 'talk.religion.misc')
                  & (test_df['Predicted Name'] == 'soc.religion.christian')]
         .sort_values(by=['Predicted Confidence'], ascending=False).head(5))
res_df
```

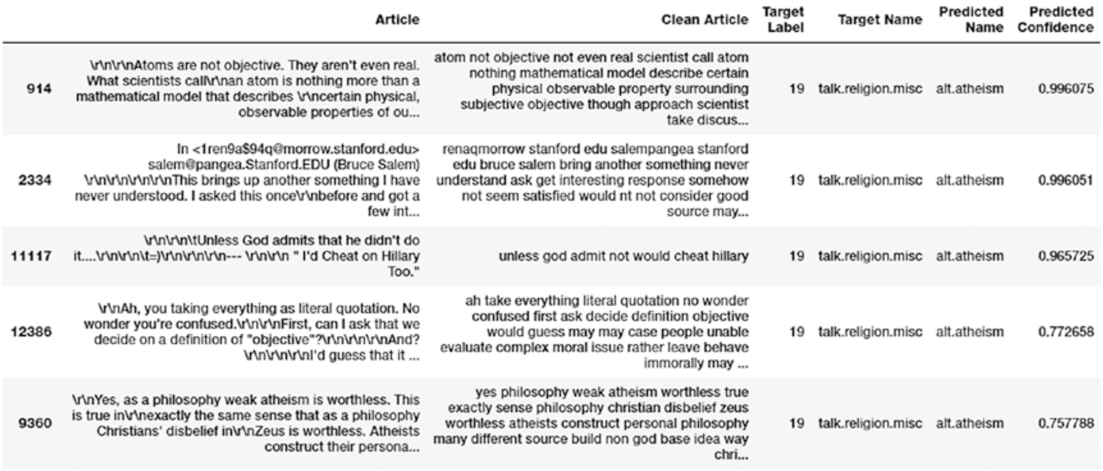


	Article	Clean Article	Target Label	Target Name	Predicted Name	Predicted Confidence
8968	\r\nZoroaster is far older than Daniel. If anything, one could claim that,\r\nin a sense, Daniel is a descendant of Zoroaster; as Daniel, though being\r\nHebrew, has assimilated into Zoroastrianis...	zoroaster far old daniel anything one could claim sense daniel descendant zoroaster daniel though hebrew assimilate zoroastrianism successfully introduce religion tanakh judaism however majority b...	19	talk.religion.misc	soc.religion.christian	0.999557
3299	There were some recent developments in the dispute about Masonry among\r\nSouthern Baptists. I posted a summary over in bit.listserv.christia, and\r\nI suppose that it might be useful here. Note...	recent development dispute masonry among southern baptists post summary bit listserv christia suppose may useful note not necessarily agree disagree follow present information short summary southe...	19	talk.religion.misc	soc.religion.christian	0.999384
4367	:\r\n (lots of stuff about the Nicene Creed deleted which can be read in the\r\n original basenote. I will also leave it up to other LDS netters to\r\n take Mr. Weiss to task on using Mormon Do...	lot stuff nicene creed delete read original basenote also leave lds netter take mr weiss task use mormon doctrine declare difinitive word lds church teach doctrine hopefully lds netter amiable exp...	19	talk.religion.misc	soc.religion.christian	0.999254
12608	: >: >> Gilligan = Sloth\r\n: >: >> Skipper = Anger\r\n: >: >> Thurston Howell III = Greed\r\n: >: >> Lovey Howell = Gluttony\r\n: >: >> Ginger = Lust\r\n: >: >> Professor = Pride\r\n: >: >> Mary ...	gilligan sloth skipper anger thurston howell iii greed lovey howell gluttony ginger lust professor pride mary ann envy assorted monkeys secular humanism assorted headhunters godless heathen savage...	19	talk.religion.misc	soc.religion.christian	0.998755
16758	\r\n\r\nThere were many injustices in the middle ages. And this is truely sad.\r\nI would hate to see a day when churches put people to death or torchured\r\nthem for practicing homosexuality, or...	many injustice middle age truely sad would hate see day church put people death torchur practice homosexuality crime church not call take government world may homosexual treat cruelly today not me...	19	talk.religion.misc	soc.religion.christian	0.998243

Figure 5-16. *Looking at mode misclassification instances for religion.misc and religion.christian*

This should enable you to take a deep dive into which instances might be getting misclassified and why. It looks like there are definitely some aspects of Christianity also mentioned in some of these articles, which leads the model to predict the `soc.religion.christian` category. Let's now take a look at some articles that were of the newsgroup `talk.religion.misc` but our model predicted `alt.atheism` with the highest confidence. See Figure 5-17.

```
pd.set_option('display.max_colwidth', 200)
res_df = (test_df[(test_df['Target Name'] == 'talk.religion.misc')
                  & (test_df['Predicted Name'] == 'alt.atheism')]
          .sort_values(by=['Predicted Confidence'], ascending=False).
          head(5))
res_df
```



	Article	Clean Article	Target Label	Target Name	Predicted Name	Predicted Confidence
914	\r\n\r\nAtoms are not objective. They aren't even real. What scientists call\r\nan atom is nothing more than a mathematical model that describes \r\ncertain physical, observable properties of ou...	atom not objective not even real scientist call atom nothing mathematical model describe certain physical observable property surrounding subjective objective though approach scientist take discus...	19	talk.religion.misc	alt.atheism	0.996075
2334	In <1ren9a$94q@morrow.stanford.edu> salem@pangea.Stanford.EDU (Bruce Salem) \r\n\r\n\r\n\r\nThis brings up another something I have never understood. I asked this once\r\nbefore and got a few int...	renaqmorrow stanford edu salempangea stanford edu bruce salem bring another something never understand ask get interesting response somehow not seem satisfied would nt not consider good source may...	19	talk.religion.misc	alt.atheism	0.996051
11117	\r\n\r\n\tUnless God admits that he didn't do it....\r\n\r\n\t=)\r\n\r\n\r\n--- \r\n\r\n " I'd Cheat on Hillary Too."	unless god admit not would cheat hillary	19	talk.religion.misc	alt.atheism	0.965725
12386	\r\nAh, you taking everything as literal quotation. No wonder you're confused.\r\n\r\nFirst, can I ask that we decide on a definition of "objective"?\r\n\r\n\r\nAnd? \r\n\r\n\r\nI'd guess that it ...	ah take everything literal quotation no wonder confused first ask decide definition objective would guess may may case people unable evaluate complex moral issue rather leave behave immorally may ...	19	talk.religion.misc	alt.atheism	0.772658
9360	\r\nYes, as a philosophy weak atheism is worthless. This is true in\r\nexactly the same sense that as a philosophy Christians' disbelief in\r\nZeus is worthless. Atheists construct their persona...	yes philosophy weak atheism worthless true exactly sense philosophy christian disbelief zeus worthless atheists construct personal philosophy many different source build non god base idea way chri...	19	talk.religion.misc	alt.atheism	0.757788

***Figure 5-17.** Looking at mode misclassification instances for religion.misc and alt.atheism*

This should be a no-brainer considering atheism and religion are related in several aspects when people talk about them, especially on online forums. Do you notice any other interesting patterns? Go ahead and explore the data further! This brings us to the end of our discussion and implementation of our text classification system. Feel free to implement more models using other innovative feature extraction techniques or supervised learning algorithms and compare their performance.

Applications

Text classification and categorization are used in several real-world scenarios and applications. Some of them are as follows:

- News categorization
- Spam filtering
- Music or movie genre categorization
- Sentiment analysis
- Language detection

The possibilities with text data are indeed endless and you can apply classification to solve various problems and automate otherwise time-consuming operations and scenarios with a little bit of effort.

Summary

Text classification is indeed a powerful tool and we have covered almost all aspects related to it in this chapter. We started off our journey with look at the definition and scope of text classification. Next, we defined automated text classification as a supervised learning problem and looked at the various types of text classification. We also briefly covered some machine learning concepts related to the various types of algorithms. A typical text classification system blueprint was also defined to describe the various modules and steps involved when building an end-to-end text classifier. Each module in the blueprint was then expanded upon.

Text preprocessing and normalization was touched upon in detail in the previous chapter and we built a normalization module especially for text classification. We saw a brief but detailed recap of various feature extraction and engineering techniques for text data from Chapter 4, including Bag of Words, TF-IDF, and advanced word embedding techniques. You should now be clear about not only the mathematical representations and concepts but also ways to implement them.

Various supervised learning methods were discussed with focus on state-of-the-art classifiers like multinomial Naïve Bayes, logistic regression, support vector machines, and ensemble models like random forest and gradient boosting. We even took a glimpse of a neural network model! We also looked at ways to evaluate classification model performance and even implemented those metrics. Finally, we put everything we learned together into building a robust 20-class text classification system on real data and evaluated various models and analyzed model performance in detail. We wrapped up our discussion by looking at some areas where text classification is used frequently. We have just scratched the surface of text analytics and NLP and we look at more ways to analyze and derive insights from textual data in the future chapters.

CHAPTER 6

Text Summarization and Topic Models

We have come quite a long way in our journey through the world of text analytics and natural language processing. You have seen how to process and annotate textual data for various applications. We also looked at state-of-the-art text representation methods with feature engineering. We also ventured into the world of machine learning and built our own multi-class text classification system by leveraging various feature extraction techniques and supervised machine learning algorithms. In this chapter, we tackle a slightly different problem in the world of text analytics—information summarization.

The world is rapidly evolving with regard to technology, commerce, business, and media. Gone are the days when we would wait for newspapers to come to our home so we could be updated about the various events around the world. With the advent of the Internet and social media, we have ushered in the so-called Information Age. Now we have various forms of social media that we consume to stay updated about daily events and stay connected with the world and our friends and family. Social media like Facebook and Twitter have created a completely different dimension to sharing and consuming information with very short messages or statuses. Humans tend to have short attention spans and this leads to us getting bored when reading large text documents and articles.

This brings us to *text summarization,* which is an extremely important concept in text analytics. It's used by businesses as well as analytical firms to shorten and summarize huge documents so that they retain the key theme of the document. Usually we present this summarized information to consumers and clients so they can understand this information in a matter of seconds. This is analogous to an elevator pitch, where you need to provide a quick summary that describes a process, product, service, or business, ensuring that it retains the core important themes and values.

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_6

This originates from the idea that the pitch should take the time it usually takes to ride an elevator, which ranges from a few seconds to a couple of minutes.

Imagine that you have a whole corpus of text documents and are tasked with deriving meaningful insights from them. At the first glance, it might seem difficult because you do not even know what to do with these documents, let alone how to apply NLP or data science to them. Since it is more about pattern mining than predictive analytics, a good way to start is to use unsupervised learning methods specifically aimed at text summarization and information extraction. In general, there are several operations that can be executed on text documents, as follows:

- **Key-phrase extraction:** This focuses on extracting key influential phrases from the documents.
- **Topic modeling:** Extract various diverse concepts or topics present in the documents, retaining the major themes in these documents.
- **Document summarization:** Summarize entire text documents to provide a gist that retains the important parts of the whole corpus.

We cover essential concepts, techniques, and practical implementations of all the three major techniques.

In this chapter, we start with a detailed discussion of the various types of summarization and information-extraction techniques and cover some foundational concepts essential for understanding the practical hands-on examples later. We cover three major techniques, including key-phrase extraction, topic models, and automated document summarization. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at `https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition`.

Text Summarization and Information Extraction

Text summarization and information extraction deals with trying to extract key concepts and themes from a huge corpus of text, essentially reducing it in the process. Before we dive deeper into the concepts and techniques, we should first understand the need for text summarization. The concept of information overload is one of the prime reasons behind the demand for text summarization. Since print and verbal media came into prominence, there has been an abundance of books, articles, audio, and video. This started around the 3rd or 4th Century BC, when people referred to the huge quantity of books as, “there was no end to the production of books” and this overload of information was often met with disapproval.

The Renaissance gave us the invention of the printing press by Gutenberg in around 1440 AD and this led to a mass production of books, manuscripts, articles, and pamphlets. This caused information overload again, with scholars complaining about excess of information, which was becoming extremely difficult to consume, process, and manage.

With the advances in computers and technology, we ushered into the digital age in the 20th Century, which gifted us with the Internet. This opened up a whole window of possibilities, into producing and consuming information with social media, news websites, electronic mail, and instant messaging capabilities. This has led to an increase in information and even led to unwanted information in the form of spam, unwanted statuses, tweets, and even bots that post unwanted content across the web.

Now that we know the current state of information being produced and consumed, we can define *information overload* as the presence of excess data or information that leads to consumers having difficulty in processing that information and making well informed decisions. This overload occurs when the amount of information as input to the system starts exceeding the processing capability of the system. Humans have limited cognitive processing capabilities and are also wired in such a way that we cannot spend a long time in reading a single piece of information or data since the mind tends to wander every now and then. Thus, when we get loaded with information, it leads to a reduction in making qualitative decisions.

Businesses thrive on making well informed decisions and they usually have a huge amount of data and information. Getting insights from this information is no piece of cake and automating it is tough because you need to know what to do with the data. Executives rarely have time to listen to long talks or go through pages of information regarding important information. The goal of summarization and information extraction is to get an idea of the key important topics and themes and summarize huge documents of information into a few lines that can be read, understood, and interpreted. The end goal is to be able to make well informed decisions in shorter timeframes. We need efficient and scalable processes and techniques that can perform this on text data. The most popular techniques are as follows:

- Key-phrase extraction
- Topic modeling
- Automated document summarization

The first two techniques involve extracting key information in the form of concepts, topics, and themes from documents, thus reducing them. Automated document summarization is all about summarizing large text documents into a few lines that explain the information the document is trying to convey. We cover each technique in detail in future sections, along with practical examples. First, we briefly talk about what each technique entails and their scope.

Keyphrase Extraction

This is perhaps the most simple out of the three techniques. It involves the process of extracting keywords or phrases from a text document or corpus that capture the main concepts or themes from the document or corpus. This can be said to be a simplistic form of topic modeling. You might have seen keywords or phrases described in a research paper or even some product in an online store that describe the entity in a few words or phrases capturing the main idea or concept of the entity.

Topic Modeling

This usually involves using statistical and mathematical modeling techniques to extract main topics, themes, or concepts from a corpus of documents. Note that we emphasize a corpus of documents because the more diverse set of documents you have, the more topics or concepts you can generate. This is unlike a single document, where you will not get too many topics or concepts if it talks about a singular concept. Topic models are also often known as *probabilistic statistical models* and they use specific statistical techniques including Singular Value Decomposition (SVD) and Latent Dirichlet Allocation (LDA) to discover connected latent semantic structures in text data to yield topics and concepts. They are used extensively in text analytics and across diverse domains like bioinformatics.

Automated Document Summarization

This is the process of using a computer program or algorithm based on statistical and machine learning techniques to summarize a document or a corpus of documents in order to obtain a short summary that captures its essential concepts and themes. A wide variety of techniques for building automated document summarizers exist, including various extraction and abstraction based techniques. The key concept behind all these algorithms is to find a representative subset of the original dataset so that the core essence of the dataset from the semantic and conceptual standpoints is contained in this subset. Document summarization usually involves extracting and constructing an executive summary from a single document, but the same algorithms can be extended to multiple documents. However, the idea is not to combine several diverse documents together because that would defeat the purpose of the algorithm. The same concept is applied to image and video summarization as well.

We now discuss some important foundational concepts around math and machine learning before moving to each technique in further detail.

Important Concepts

There are several important mathematical and machine learning foundational concepts that we discuss in this section that will be useful later. Some of these will be familiar to you, but we will repeat them for the sake of completeness so that you can refresh your memory. We also cover some concepts from natural language processing in this section.

A *document* is an entity containing a whole body of text data with optional headers and other metadata information. A *corpus* usually consists of a collection of documents. These documents can be simple sentences or complete paragraphs of textual information. A *tokenized corpus* refers to a corpus where each document is tokenized or broken down into tokens, which are usually words.

Text wrangling or *preprocessing* is the process of cleaning, normalizing, and standardizing textual data with techniques like removing special symbols and characters, removing extraneous HTML tags, removing stopwords, correcting spellings, stemming, and lemmatization.

Feature engineering is a process where we extract meaningful feature or attributes from raw textual data and feed it into a statistical or machine learning algorithm. This process is also known as *vectorization* since the end transformation of this process is numerical vectors from raw text tokens. The reason is that conventional algorithms work on numerical vectors and cannot work directly on raw text data. There are various feature extraction methods, including Bag of Words based binary features, which tell us if a word or group of words exist in the document, Bag of Words based frequency features that tell us the frequency of a word or group of words in a document, and term frequency-inverse document frequency or TF-IDF weighted features, which take into account the term frequency and inverse document frequency when weighing each term. You can look at Chapter 4 for more details on feature extraction.

A *feature matrix* usually refers to a mapping from a collection of documents to features where each row indicates a document and each column indicates a particular feature (usually a word or a set of words). We represent a collection of documents or sentences through feature matrices after feature extraction and we will often apply statistical and machine learning techniques on these matrices in our practical examples. A feature matrix can also be transposed where, instead of a conventional document-term matrix, we end up with a term-document matrix. We can also represent other features like document similarity, topic-terms, and topic-documents as feature matrices.

Singular Value Decomposition (SVD) is a technique from linear algebra that's used quite frequently in summarization algorithms. SVD is the process of factorization of a matrix that is real or complex. Formally, we can define SVD as follows. Consider a matrix **M** which has dimensions of $m \times n$, where m denotes the number of rows and n denotes the number of columns. Mathematically, the matrix **M** can be represented using SVD as a factorization such that

$$M_{m\times n} = U_{m\times m}\, S_{m\times n}\, V^{T}_{n\times n}$$

where we have the following decompositions:

- **U** is a $m \times m$ unitary matrix such that $U^T U = I_{m \times m}$ where **I** is the identity matrix. The columns of **U** indicate left singular vectors.
- **S** is a diagonal $m \times n$ matrix with positive real numbers on the diagonal of the matrix. This is also often also represented as a vector of m values which indicate the singular values.
- $\boldsymbol{V^T}$ is a $n \times n$ unitary matrix such that $V^T V = I_{n \times n}$, where **I** is the identity matrix. The rows of **V** indicate right singular vectors.

This tells us that **U** and **V** are orthogonal. The singular values of **S** are particularly important in summarization algorithms. We use SVD particularly for low rank matrix approximation, where we approximate the original matrix **M** with a matrix $\hat{M}$ such that this new matrix is a truncated version of the original matrix **M** with a rank **k** and can be represented by SVD as $\hat{M} = U\hat{S}V^T$, where $\hat{S}$ is a truncated version of the original ***S*** matrix and now consists of only the top **k** largest singular values and the other singular values are represented by zero. We use a nice implementation from SciPy to extract the top **k** singular values and return the corresponding **U**, **S**, and **V** matrices. The code snippet we use is in the `utils.py` file and is depicted here:

```
from scipy.sparse.linalg import svds

def low_rank_svd(matrix, singular_count=2):

    u, s, vt = svds(matrix, k=singular_count)
    return u, s, vt
```

We use this implementation in topic modeling as well as document summarization in future sections. Figure 6-1 shows a nice depiction of this process, which yields ***k*** singular vectors from the original SVD decomposition and shows how we can determine the low rank matrix approximation.

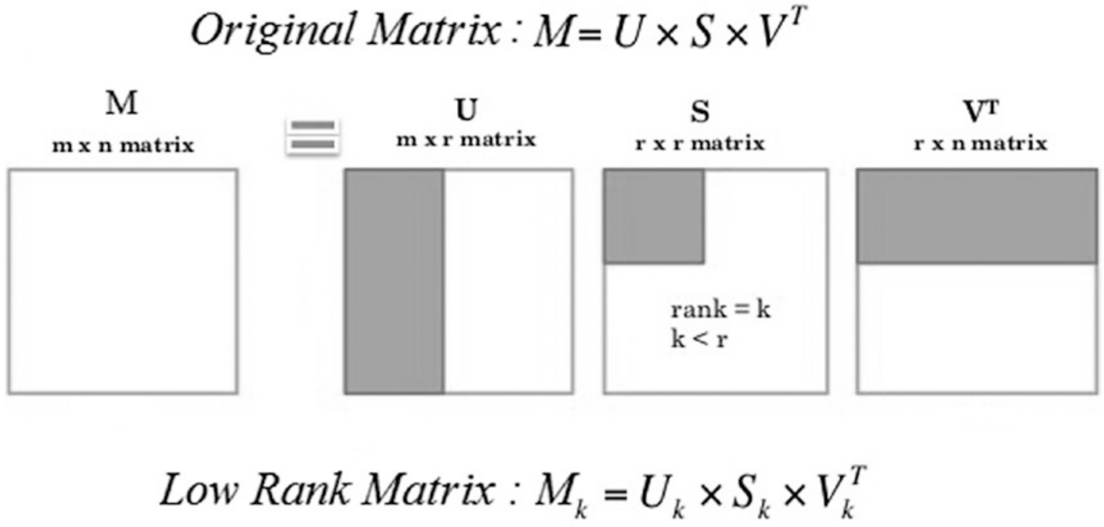

Original Matrix : $M = U \times S \times V^T$
M
m x n matrix
U
m x r matrix
S
r x r matrix
VT
r x n matrix
rank = k
k < r
Low Rank Matrix : $M_k = U_k \times S_k \times V_k^T$

Figure 6-1. *Singular Value Decomposition with low rank matrix approximation*

From Figure 6-1, you can clearly see that **k** singular values are retained in the low rank matrix approximation and how the original matrix **M** is decomposed into **U**, **S**, and **VT** using SVD.

- **M** is typically known as the *term-document* matrix and is usually obtained after feature engineering on the preprocessed text data, where each row of the matrix represents a term and each column represents a text document.

- **U** is known as the *term-topic* matrix where each row of the matrix represents a term and each column represents a topic. It's useful for getting the influential terms for each topic when we multiply this by the singular values.
- **S** is the matrix or array that consists of the list of *singular values* obtained after low-rank SVD, which is typically equal to the number of topics we decide prior to this operation.
- **VT** is the *topic-document* matrix, which if you transpose, you get the *document-topic* matrix, which is useful in knowing how much influence each topic has on each document.

We try to keep the math to a minimum in the rest of the chapter unless it is absolutely essential to understand how the algorithms work. But we encourage readers to dive deeper into these techniques for a better understanding of how they work behind the scenes.

Keyphrase Extraction

This is one of the simplest yet most powerful techniques of extracting important information from unstructured text documents. Keyphrase extraction, also known as *terminology extraction,* is the process of extracting key terms or phrases from a body of unstructured text so that the core themes are captured. This technique falls under the broad umbrella of information retrieval and extraction. Keyphrase extraction is useful in many areas, some of which are mentioned here:

- Semantic web
- Query based search engines and crawlers
- Recommendation systems
- Tagging systems
- Document similarity
- Translation

Keyphrase extraction is often the starting point for carrying out more complex tasks in text analytics or natural language processing and the output can act as features for more complex systems. There are various approaches for keyphrase extraction; we cover the following two major techniques:

- Collocations
- Weighted tag-based phrase extraction

An important point to remember is that we will be extracting phrases, which are usually a collection of words and can sometimes just be single words. If you are extracting keywords, that is also known as keyword extraction and it is a subset of keyphrase extraction.

Collocations

The term *collocation* is borrowed from analyzing corpora and linguistics. A collocation can be defined as a sequence or group of words that tend to occur frequently and this frequency tends to be more than what could be termed a random or chance occurrence. Various types of collocations can be formed based on parts of speech like nouns, verbs, and so on. There are various ways to extract collocations and one of the best ways to do it is to use an n-gram grouping or segmentation approach. This is where we construct n-grams out of a corpus and then count the frequency of each n-gram and rank them based on their frequency of occurrence to get the most frequent n-gram collocations.

The idea is to have a corpus of documents (paragraphs or sentences), tokenize them to form sentences, flatten the list of sentences to form one large sentence or string over which we slide a window of size n based on the n-gram range, and compute n-grams across the string. Once they are computed, we count each n-gram based on its frequency of occurrence and then rank it. This yields the most frequent collocations on the basis of frequency. We implement this from scratch initially so that you can understand the algorithm better and then we use some of NLTK's built-in capabilities to depict it.

Let' start by loading some necessary dependencies and a corpus on which we will be computing collocations. We use the NLTK Gutenberg corpus's book, Lewis Carroll's *Alice in Wonderland,* as our corpus. We also normalize the corpus to standardize the text content using our handy `text_normalizer` module, which we built and used in the previous chapters.

```
from nltk.corpus import gutenberg
import text_normalizer as tn
import nltk
from operator import itemgetter

# load corpus
alice = gutenberg.sents(fileids='carroll-alice.txt')
alice = [' '.join(ts) for ts in alice]
norm_alice = list(filter(None,
                        tn.normalize_corpus(alice, text_lemmatization=False)))

# print and compare first line
print(alice[0], '\n', norm_alice[0])

[ Alice ' s Adventures in Wonderland by Lewis Carroll 1865 ]
 alice adventures wonderland lewis carroll
```

Now we define a function to compute n-grams based on some input list of tokens and the parameter **n**, which determines the degree of the n-gram like a uni-gram, bi-gram, and so on. The following code snippet computes n-grams for an input sequence.

```
def compute_ngrams(sequence, n):
    return list(
            zip(*(sequence[index:]
                    for index in range(n)))
    )
```

This function basically takes in a sequence of tokens and computes a list of lists having sequences where each list contains all items from the previous list except the first item removed from the previous list. It constructs n such lists and then zips them all together to give us the necessary n-grams. We wrap the final result in a list since in Python 3; `zip` gives us a generator object and not a raw list. We can see the function in action on a sample sequence in the following snippet.

```
In [7]: compute_ngrams([1,2,3,4], 2)
Out[7]: [(1, 2), (2, 3), (3, 4)]

In [8]: compute_ngrams([1,2,3,4], 3)
Out[8]: [(1, 2, 3), (2, 3, 4)]
```

The preceding output shows bi-grams and tri-grams for an input sequence. We now utilize this function and build upon it to generate the top n-grams based on their frequency of occurrence. For this, we need to define a function to flatten the corpus into one big string of text. The following function help us do this on a corpus of documents.

```
def flatten_corpus(corpus):
    return ' '.join([document.strip()
                     for document in corpus])
```

We can now build a function that will help us get the top n-grams from a corpus of text.

```
def get_top_ngrams(corpus, ngram_val=1, limit=5):

    corpus = flatten_corpus(corpus)
    tokens = nltk.word_tokenize(corpus)

    ngrams = compute_ngrams(tokens, ngram_val)
    ngrams_freq_dist = nltk.FreqDist(ngrams)
    sorted_ngrams_fd = sorted(ngrams_freq_dist.items(),
                              key=itemgetter(1), reverse=True)
    sorted_ngrams = sorted_ngrams_fd[0:limit]
    sorted_ngrams = [(' '.join(text), freq)
                     for text, freq in sorted_ngrams]

    return sorted_ngrams
```

We use NLTK's `FreqDist` class to create a counter of all the n-grams based on their frequency and then we sort them based on their frequency and return the top n-grams based on the specified user limit. We now compute the top bi-grams and tri-grams on our corpus using the following code snippet.

```
# top 10 bigrams
In [11]: get_top_ngrams(corpus=norm_alice, ngram_val=2,
    ...:                limit=10)
Out[11]:
[('said alice', 123),
 ('mock turtle', 56),
 ('march hare', 31),
 ('said king', 29),
 ('thought alice', 26),
 ('white rabbit', 22),
 ('said hatter', 22),
 ('said mock', 20),
 ('said caterpillar', 18),
 ('said gryphon', 18)]

# top 10 trigrams
In [12]: get_top_ngrams(corpus=norm_alice, ngram_val=3,
     ...:               limit=10)
Out[12]:
[('said mock turtle', 20),
 ('said march hare', 10),
 ('poor little thing', 6),
 ('little golden key', 5),
 ('certainly said alice', 5),
 ('white kid gloves', 5),
 ('march hare said', 5),
 ('mock turtle said', 5),
 ('know said alice', 4),
 ('might well say', 4)]
```

This output shows us sequences of two and three words generated by n-grams along with the number of times they occur throughout the corpus. We can see that most of the collocations point to people who are speaking something as "said <person>". We also see the people who are popular characters in Alice in Wonderland, like the mock turtle, the king, the rabbit, the hatter, and Alice are depicted in the collocations.

We now look at NLTK's collocation finders, which enable us to find collocations using various measures like raw frequencies, pointwise mutual information, and so on. Just to explain briefly, *pointwise mutual information* can be computed for two events or terms as the logarithm of the ratio of the probability of them occurring together by the product of their individual probabilities, assuming that they are independent of each other. Mathematically, we can represent it as follows:

$$pmi(x,y) = \log \frac{p(x,y)}{p(x)p(y)}$$

This measure is symmetric. The following code snippet shows us how to compute these collocations using these measures.

```
# bigrams
from nltk.collocations import BigramCollocationFinder
from nltk.collocations import BigramAssocMeasures

finder = BigramCollocationFinder.from_documents([item.split()
                                                 for item
                                                 in norm_alice])
finder

<nltk.collocations.BigramCollocationFinder at 0x1c2c2c4f358>

# raw frequencies
In [14]: finder.nbest(bigram_measures.raw_freq, 10)
Out[14]:
[(u'said', u'alice'),
 (u'mock', u'turtle'),
 (u'march', u'hare'),
 (u'said', u'king'),
 (u'thought', u'alice'),
 (u'said', u'hatter'),
 (u'white', u'rabbit'),
 (u'said', u'mock'),
 (u'said', u'caterpillar'),
 (u'said', u'gryphon')]

# pointwise mutual information
In [15]: finder.nbest(bigram_measures.pmi, 10)
Out[15]:
[(u'abide', u'figures'),
 (u'acceptance', u'elegant'),
 (u'accounting', u'tastes'),
 (u'accustomed', u'usurpation'),
 (u'act', u'crawling'),
 (u'adjourn', u'immediate'),
 (u'adoption', u'energetic'),
 (u'affair', u'trusts'),
 (u'agony', u'terror'),
 (u'alarmed', u'proposal')]

# trigrams
from nltk.collocations import TrigramCollocationFinder
from nltk.collocations import TrigramAssocMeasures

finder = TrigramCollocationFinder.from_documents([item.split()
                                                  for item
                                                  in norm_alice])
trigram_measures = TrigramAssocMeasures()

# raw frequencies
In [17]: finder.nbest(trigram_measures.raw_freq, 10)
Out[17]:
[(u'said', u'mock', u'turtle'),
 (u'said', u'march', u'hare'),
 (u'poor', u'little', u'thing'),
 (u'little', u'golden', u'key'),
 (u'march', u'hare', u'said'),
 (u'mock', u'turtle', u'said'),
 (u'white', u'kid', u'gloves'),
 (u'beau', u'ootiful', u'soo'),
 (u'certainly', u'said', u'alice'),
 (u'might', u'well', u'say')]

# pointwise mutual information
In [18]: finder.nbest(trigram_measures.pmi, 10)
Out[18]:
[(u'accustomed', u'usurpation', u'conquest'),
 (u'adjourn', u'immediate', u'adoption'),
 (u'adoption', u'energetic', u'remedies'),
 (u'ancient', u'modern', u'seaography'),
 (u'apple', u'roast', u'turkey'),
 (u'arithmetic', u'ambition', u'distraction'),
 (u'brother', u'latin', u'grammar'),
 (u'canvas', u'bag', u'tied'),
 (u'cherry', u'tart', u'custard'),
 (u'circle', u'exact', u'shape')]
```

Now you know how to compute collocations for a corpus using an n-gram generative approach. We look at a better way of generating keyphrases based on parts of speech (PoS) tagging and term weighing in the next section.

Weighted Tag-Based Phrase Extraction

We now look at a slightly different approach to extracting keyphrases. This method borrows concepts from a couple of papers, namely K. Barker and N. Cornachhia's "Using Noun Phrase Heads to Extract Document Keyphrases" and Ian Witten et al.'s "KEA: Practical Automatic Keyphrase Extraction," which you can refer to if you are interested in further details on their experimentations and approaches. We follow a two-step process in our algorithm, as follows:

1. Extract all noun phrase chunks using shallow parsing.
2. Compute TF-IDF weights for each chunk and return the top weighted phrases.

For the first step, we use a simple pattern based on parts of speech (POS) tags to extract noun phrase chunks. You will be familiar with this from Chapter 3, where we explored chunking and shallow parsing. Before discussing our algorithm, let's load the corpus on which we will be testing our implementation. We use a sample description of elephants taken from Wikipedia, available in the `elephants.txt` file, which you can obtain from the GitHub repository for this book at https://github.com/dipanjanS/text-analytics-with-python.

```
data = open('elephants.txt', 'r+').readlines()
sentences = nltk.sent_tokenize(data[0])
len(sentences)
```

```
29
```

```
# viewing the first three lines
sentences[:3]
```

```
['Elephants are large mammals of the family Elephantidae and the order
Proboscidea.', 'Three species are currently recognised: the African bush
elephant (Loxodonta africana), the African forest elephant (L. cyclotis),
and the Asian elephant (Elephas maximus).', 'Elephants are scattered
throughout sub-Saharan Africa, South Asia, and Southeast Asia.']
```

Let's now use our nifty text_normalizer module to do some very basic text preprocessing on our corpus.

```
norm_sentences = tn.normalize_corpus(sentences, text_lower_case=False,
                                    text_stemming=False, text_
                                    lemmatization=False,
                                    stopword_removal=False)
norm_sentences[:3]
```

```
['Elephants are large mammals of the family Elephantidae and the order
Proboscidea', 'Three species are currently recognised the African bush
elephant Loxodonta africana the African forest elephant L cyclotis and
the Asian elephant Elephas maximus', 'Elephants are scattered throughout
subSaharan Africa South Asia and Southeast Asia']
```

Now that we have our corpus ready, we will use the pattern " NP: {<DT>? <JJ>* <NN.*>+}" to extract all possible noun phrases from our corpus of documents/sentences. You can always experiment with more sophisticated patterns, later incorporating verb, adjective, or even adverb phrases. However, we keep things simple and concise here to focus on the core logic. Once we have our pattern, we will define a function to parse and extract these phrases using the following snippet. We also load any other necessary dependencies at this point.

```
import itertools
stopwords = nltk.corpus.stopwords.words('english')

def get_chunks(sentences, grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
               stopword_list=stopwords):

    all_chunks = []
    chunker = nltk.chunk.regexp.RegexpParser(grammar)
    for sentence in sentences:

        tagged_sents = [nltk.pos_tag(nltk.word_tokenize(sentence))]

        chunks = [chunker.parse(tagged_sent)
                      for tagged_sent in tagged_sents]

        wtc_sents = [nltk.chunk.tree2conlltags(chunk)
                        for chunk in chunks]

        flattened_chunks = list(
                            itertools.chain.from_iterable(
                                wtc_sent for wtc_sent in wtc_sents)
                           )

        valid_chunks_tagged = [(status, [wtc for wtc in chunk])
                                    for status, chunk
                                        in itertools.groupby(flattened_chunks,
                                        lambda word_pos_chunk: word_pos_
                                        chunk[2] != 'O')]

        valid_chunks = [' '.join(word.lower()
                                 for word, tag, chunk in wtc_group
                                     if word.lower() not in stopword_list)
                                         for status, wtc_group in valid_
                                         chunks_tagged
                                             if status]

        all_chunks.append(valid_chunks)

    return all_chunks
```

In this function, we have a defined grammar pattern for chunking or extracting noun phrases. We define a chunker over the same pattern and, for each sentence in the document, we first annotate it with its POS tags and then build a shallow parse tree with noun phrases as the chunks and all other POS tag based words as chinks, which are not parts of any chunks. Once this is done, we use the `tree2conlltags` function to generate `(w,t,c)` *triples*, which are words, POS tags, and the IOB formatted chunk tags (discussed in Chapter 3). We remove all tags with a chunk tag of `O`, since they are basically words or terms that do not belong to any chunk. Finally, from these valid chunks, we combine the chunked terms to generate phrases from each chunk group. We can see this function in action on our corpus in the following snippet.

```
chunks = get_chunks(norm_sentences)
chunks
```

```
[['elephants', 'large mammals', 'family elephantidae', 'order
proboscidea'],
 ['species', 'african bush elephant loxodonta', 'african forest elephant l
  cyclotis', 'asian elephant elephas maximus'],
 ['elephants', 'subsaharan africa south asia', 'southeast asia'],
...,
...,
 ['incisors', 'tusks', 'weapons', 'tools', 'objects'],
 ['elephants', 'flaps', 'body temperature'],
 ['pillarlike legs', 'great weight'],
...,
...,
 ['threats', 'populations', 'ivory trade', 'animals', 'ivory tusks'],
 ['threats', 'elephants', 'habitat destruction', 'conflicts', 'local people'],
 ['elephants', 'animals', 'asia'],
 ['past', 'war today', 'display', 'zoos', 'entertainment', 'circuses'],
 ['elephants', 'art folklore religion literature', 'popular culture']]
```

This output shows us all the valid keyphrases per sentence of our document. You can already see since we targeted noun phrases, all phrases talk about noun based entities. We now build on top of our `get_chunks()` function by implementing the necessary logic for Step 2, where we will build a TF-IDF based model on our keyphrases using Gensim and then compute TF-IDF based weights for each keyphrase based on its occurrence in the corpus. Finally, we sort these keyphrases based on their TF-IDF weights and show the top **N** keyphrases, where `top_n` is specified by the user.

```
from gensim import corpora, models

def get_tfidf_weighted_keyphrases(sentences,
                                  grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
                                  top_n=10):

    valid_chunks = get_chunks(sentences, grammar=grammar)

    dictionary = corpora.Dictionary(valid_chunks)
    corpus = [dictionary.doc2bow(chunk) for chunk in valid_chunks]

    tfidf = models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]

    weighted_phrases = {dictionary.get(idx): value
                           for doc in corpus_tfidf
                               for idx, value in doc}

    weighted_phrases = sorted(weighted_phrases.items(),
                              key=itemgetter(1), reverse=True)
    weighted_phrases = [(term, round(wt, 3)) for term, wt in weighted_phrases]

    return weighted_phrases[:top_n]
```

We can now test this function on our toy corpus by using the following code snippet to generate the top 30 keyphrases.

```
# top 30 tf-idf weighted keyphrases
get_tfidf_weighted_keyphrases(sentences=norm_sentences, top_n=30)
```

```
[('water', 1.0), ('asia', 0.807), ('wild', 0.764), ('great weight', 0.707),
 ('pillarlike legs', 0.707), ('southeast asia', 0.693), ('subsaharan africa
 south asia', 0.693), ('body temperature', 0.693), ('flaps', 0.693),
 ('fissionfusion society', 0.693), ('multiple family groups', 0.693),
 ('art folklore religion literature', 0.693), ('popular culture', 0.693),
 ('ears', 0.681), ('males', 0.653), ('males bulls', 0.653), ('family
 elephantidae', 0.607), ('large mammals', 0.607), ('years', 0.607),
 ('environments', 0.577), ('impact', 0.577), ('keystone species', 0.577),
 ('cetaceans', 0.577), ('elephant intelligence', 0.577), ('primates',
 0.577), ('dead individuals', 0.577), ('kind', 0.577), ('selfawareness',
 0.577), ('different habitats', 0.57), ('marshes', 0.57)]
```

Interestingly, we see various types of elephants being depicted in the keyphrases like Asian and African elephants and typical attributes of elephants like “great weight”, “fission fusion society” and “pillar like legs”.

We can also leverage Gensim’s summarization module, which has a keywords function that extracts keywords from the text. This uses a variation of the TextRank algorithm, which we explore in the document summarization section.

```
from gensim.summarization import keywords
```

```
key_words = keywords(data[0], ratio=1.0, scores=True, lemmatize=True)
[(item, round(score, 3)) for item, score in key_words][:25]
```

```
[('african bush elephant', 0.261), ('including', 0.141), ('family', 0.137),
 ('cow', 0.124), ('forests', 0.108), ('female', 0.103), ('asia', 0.102),
 ('objects', 0.098), ('tigers', 0.098), ('sight', 0.098), ('ivory', 0.098),
 ('males', 0.088), ('folklore', 0.087), ('known', 0.087), ('religion', 0.087),
 ('larger ears', 0.085), ('water', 0.075), ('highly recognisable', 0.075),
 ('breathing lifting', 0.074), ('flaps', 0.073), ('africa', 0.072),
 ('gomphotheres', 0.072), ('animals tend', 0.071), ('success', 0.071),
 ('south', 0.07)]
```

Thus, you can see how keyphrase extraction can extract key important concepts from text documents and summarize them. Try these functions on other corpora to see interesting results!

Topic Modeling

We have seen how keyphrases can be extracted using a couple of techniques. While these phrases point out key pivotal points from a document or corpus, this technique is simplistic and often does not accurately portray the various themes or concepts in a corpus, particularly when we have different distinguishing themes or concepts in a corpus of documents.

Topic models have been designed specifically for the purpose of extracting various distinguishing concepts or topics from a large corpus that has various types of documents and each document talks about one or more concepts. These concepts can be anything from thoughts, opinions, facts, outlooks, statements, and so on. The main aim of topic modeling is to use mathematical and statistical techniques to discover hidden and latent semantic structures in a corpus.

Topic modeling involves extracting features from document terms and using mathematical structures and frameworks like matrix factorization and SVD to generate clusters or groups of terms that are distinguishable from each other and these cluster of words form topics or concepts. These concepts can be used to interpret the main themes of a corpus and make semantic connections among words that co-occur frequently in various documents. There are various frameworks and algorithms to build topic models. We cover the following three methods:

- Latent Semantic Indexing
- Latent Dirichlet Allocation
- Non-negative matrix factorization

The first two methods are quite popular and long-standing. The last technique, non-negative matrix factorization, is a recent but extremely effective technique and gives excellent results. We leverage Gensim and Scikit-Learn for our practical implementations and look at how to build our own topic model based on Latent Semantic Indexing.

We do things a bit differently in this new edition of the book in contrast to the previous edition. Instead of working on toy datasets, we work on a complex real-world dataset just like we have been doing in the other chapters. In the next few sections, we demonstrate how to perform topic modeling with the three methods mentioned previously. For demonstration, we leverage the two most popular frameworks—Gensim and Scikit-Learn. The intent here is to understand how to leverage these frameworks easily to build topic models and to understand some of the essential concepts behind the scenes.

Topic Modeling on Research Papers

We will do an interesting exercise here—build topic models on past research papers from the very popular NIPS conference (now known as the NeurIPS conference). The late professor Sam Roweis compiled an excellent collection of NIPS Conference Papers from Volume 1 – 12, which you can find at `https://cs.nyu.edu/~roweis/data.html`. An interesting fact is that he obtained this by massaging the OCR'd data from NIPS 1-12, which was actually the pre-electronic submission era. Yann LeCun made the data available. There is an even more updated dataset available up to NIPS 17 at `http://ai.stanford.edu/~gal/data.html`. However, that dataset is in the form of a MAT file, so you might need to do some additional preprocessing before working on it in Python.

The Main Objective

Considering our discussion so far, our main objective is pretty simple. Given a whole bunch of conference research papers, can we identify some key themes or topics from these papers by leveraging unsupervised learning? We do not have the liberty of labeled categories telling us what the major themes of every research paper are. Besides that, we are dealing with text data extracted using OCR (optical character recognition). Hence, you can expect misspelled words, words with characters missing, and so on, which makes our problem even more challenging. The key objectives of our topic modeling exercise are to showcase the following:

- Topic modeling with Gensim and Scikit-Learn
- Implementing topic models, using LDA, LSI, and NMF
- How to leverage third-party modeling frameworks like MALLET for topic models
- Evaluating topic modeling performance
- Tuning topic models for optimal topics
- Interpreting topic modeling results
- Predicting topics for new research papers

The bottom line is that we can identify some major themes from NIPS research papers using topic modeling, interpret these topics, and even predict topics for new research papers!

Data Retrieval

We need to retrieve the dataset available on the web. You can even download it directly from the Jupyter notebook using the following command (or from the terminal by removing the exclamation mark at the beginning of the command).

```
!wget https://cs.nyu.edu/~roweis/data/nips12raw_str602.tgz
```

```
--2018-11-07 18:59:33--  https://cs.nyu.edu/~roweis/data/nips12raw_str602.tgz
Resolving cs.nyu.edu (cs.nyu.edu)... 128.122.49.30
Connecting to cs.nyu.edu (cs.nyu.edu)|128.122.49.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12851423 (12M) [application/x-gzip]
Saving to: 'nips12raw_str602.tgz'

nips12raw_str602.tg 100%[===================>]  12.26M  1.75MB/s    in 7.4s

2018-11-07 18:59:41 (1.65 MB/s) - 'nips12raw_str602.tgz' saved
[12851423/12851423]
```

Once the archive is downloaded, you can extract the contents from it automatically by using the following command directly from the notebook.

```
!tar -xzf nips12raw_str602.tgz
```

If you are using the Windows operating system, these commands might not work and you can simply obtain the research papers manually by going to the website at https://cs.nyu.edu/~roweis/data/nips12raw_str602.tgz. Download the archive. Once it's downloaded, you can use any archive extraction tool to extract the nipstxt folder. Once the contents are extracted, we can verify them by checking out the folder structure.

```
import os
import numpy as np
import pandas as pd

DATA_PATH = 'nipstxt/'
print(os.listdir(DATA_PATH))
['nips01', 'nips04', 'MATLAB_NOTES', 'nips10', 'nips02', 'idx', 'nips11',
'nips03', 'nips07', 'README_yann', 'nips05', 'nips12', 'nips06', 'RAW_DATA_
NOTES', 'orig', 'nips00', 'nips08', 'nips09']
```

Load and View Dataset

We can now load all the research papers using the following code. Each paper is in its own text file, hence we need to use file-reading functions from Python.

```
folders = ["nips{0:02}".format(i) for i in range(0,13)]
# Read all texts into a list.
papers = []
for folder in folders:
    file_names = os.listdir(DATA_PATH + folder)
    for file_name in file_names:
        with open(DATA_PATH + folder + '/' + file_name, encoding='utf-8',
                  errors='ignore', mode='r+') as f:
            data = f.read()
        papers.append(data)
len(papers)
```

```
1740
```

There are a total of 1,740 research papers, which is not a small number! Let's take a look at a fragment of text from one of the research papers to get an idea.

```
print(papers[0][:1000])
```

```
652
Scaling Properties of Coarse-Coded Symbol Memories
Ronald Rosenfeld
David S. Touretzky
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Abstract
Coarse-coded symbol memories have appeared in several neural network
symbol processing models. In order to determine how these models would
scale, one must first have some understanding of the mathematics of coarse-
coded representations. We define the general structure of coarse-coded
```

```
symbol memories and derive mathematical relationships among their essential
parameters: memory t size, sylmbol-set size and capacitor. The computed
capacity of one of the schemes agrees well with actual measurements of
the coarse-coded working memory of DCPS, Touretzky and Hinton's distributed
connectionist production system.
1 Introduction
A distributed representation is a memory scheme in which each entity
(concept, symbol) is represented by a pattern of activity over many units
[3]. If each unit partic
```

Look at that! It's basically how a research paper looks in a PDF, which you often view in your browser. However, it looks like the OCR hasn't worked perfectly and we have some missing characters here and there. This is expected, but also makes this task more challenging!

Basic Text Wrangling

We perform some basic text wrangling or preprocessing before diving into topic modeling. We keep things simple here and perform tokenization, lemmatizing nouns, and removing stopwords and any terms having a single character.

```
%%time
import nltk

stop_words = nltk.corpus.stopwords.words('english')
wtk = nltk.tokenize.RegexpTokenizer(r'\w+')
wnl = nltk.stem.wordnet.WordNetLemmatizer()

def normalize_corpus(papers):
    norm_papers = []
    for paper in papers:
        paper = paper.lower()
        paper_tokens = [token.strip() for token in wtk.tokenize(paper)]
        paper_tokens = [wnl.lemmatize(token) for token in paper_tokens if
        not token.isnumeric()]
        paper_tokens = [token for token in paper_tokens if len(token) > 1]
        paper_tokens = [token for token in paper_tokens if token not in
        stop_words]
        paper_tokens = list(filter(None, paper_tokens))
        if paper_tokens:
            norm_papers.append(paper_tokens)

    return norm_papers

norm_papers = normalize_corpus(papers)
print(len(norm_papers))

1740
CPU times: user 38.6 s, sys: 92 ms, total: 38.7 s
Wall time: 38.7 s

# viewing a processed paper
print(norm_papers[0][:50])

['scaling', 'property', 'coarse', 'coded', 'symbol', 'memory', 'ronald',
'rosenfeld', 'david', 'touretzky', 'computer', 'science', 'department',
'carnegie', 'mellon', 'university', 'pittsburgh', 'pennsylvania',
'abstract', 'coarse', 'coded', 'symbol', 'memory', 'appeared', 'several',
'neural', 'network', 'symbol', 'processing', 'model', 'order', 'determine',
'model', 'would', 'scale', 'one', 'must', 'first', 'understanding',
'mathematics', 'coarse', 'coded', 'representa', 'tions', 'define',
'general', 'structure', 'coarse', 'coded', 'symbol']
```

We are now ready to start building topic models and will be showcasing methods in Gensim and Scikit-Learn, as mentioned earlier.

Topic Models with Gensim

The key tagline of the Gensim framework is topic modeling for humans, which makes it pretty clear that this framework was built for topic modeling. We can do amazing things with this framework, including text similarity, semantic analytics, topic models, and text summarization. Besides this, Gensim offers a lot of capabilities and more flexibility than Scikit-Learn to build, evaluate, and tune topic models, which we will see very shortly. We build topic models using the following methods in this section.

- Latent Semantic Indexing (LSI)
- Latent Dirichlet Allocation (LDA)

Without further ado, let's get started by looking at ways to generate phrases with influential bi-grams and remove some terms that may not be useful before feature engineering.

Text Representation with Feature Engineering

Before feature engineering and vectorization, we want to extract some useful bi-gram based phrases from our research papers and remove some unnecessary terms. We leverage the very useful `gensim.models.Phrases` class for this. This capability helps us automatically detect common phrases from a stream of sentences, which are typically multi-word expressions/word n-grams. This implementation draws inspiration from the famous paper by Mikolov, et al., "Distributed Representations of Words and Phrases and their Compositionality," which you can check out at https://arxiv.org/abs/1310.4546. We start by extracting and generating words and bi-grams as phrases for each tokenized research paper. We can build this phrase generation model easily with the following code and test it on a sample paper.

```
import gensim

bigram = gensim.models.Phrases(norm_papers, min_count=20, threshold=20,
delimiter=b'_') # higher threshold fewer phrases.
bigram_model = gensim.models.phrases.Phraser(bigram)

# sample demonstration
print(bigram_model[norm_papers[0]][:50])
```

```
['scaling', 'property', 'coarse_coded', 'symbol', 'memory', 'ronald',
'rosenfeld', 'david_touretzky', 'computer_science', 'department',
'carnegie_mellon', 'university_pittsburgh', 'pennsylvania', 'abstract',
'coarse_coded', 'symbol', 'memory', 'appeared', 'several', 'neural_
network', 'symbol', 'processing', 'model', 'order', 'determine', 'model',
'would', 'scale', 'one', 'must', 'first', 'understanding', 'mathematics',
'coarse_coded', 'representa_tions', 'define', 'general', 'structure',
'coarse_coded', 'symbol', 'memory', 'derive', 'mathematical',
'relationship', 'among', 'essential', 'parameter', 'memor', 'size', 'lmbol']
```

We can clearly see that we have single words as well as bi-grams (two words separated by an underscore), which tells us that our model works. We leverage the min_count parameter, which tells us that our model ignores all words and bi-grams with total collected count lower than 20 across the corpus (of the input paper as a list of tokenized sentences). We also use a threshold of 20, which tells us that the model accepts specific phrases based on this threshold value so that a phrase of words a followed by b is accepted if the score of the phrase is greater than the threshold of 20. This threshold is dependent on the scoring parameter, which helps us understand how these phrases are scored to understand their influence.

Typically the default scorer is used and it's pretty straightforward to understand. You can check out further details in the documentation at https://radimrehurek.com/gensim/models/phrases.html#gensim.models.phrases.original_scorer and in the previously mentioned research paper.

Let's generate phrases for all our tokenized research papers and build a vocabulary that will help us obtain a unique term/phrase to number mapping (since machine or deep learning only works on numeric tensors).

```
norm_corpus_bigrams = [bigram_model[doc] for doc in norm_papers]

# Create a dictionary representation of the documents.
dictionary = gensim.corpora.Dictionary(norm_corpus_bigrams)
print('Sample word to number mappings:', list(dictionary.items())[:15])
print('Total Vocabulary Size:', len(dictionary))
```

```
Sample word to number mappings: [(0, '8a'), (1, 'abandon'), (2, 'able'),
(3, 'abo'), (4, 'abstract'), (5, 'accommodate'), (6, 'accuracy'),
(7, 'achieved'), (8, 'acknowledgment_thank'), (9, 'across'), (10, 'active'),
(11, 'activity'), (12, 'actual'), (13, 'adjusted'), (14, 'adjusting')]
Total Vocabulary Size: 78892
```

Wow! Looks like we have a lot of unique phrases in our corpus of research papers, based on the preceding output. Several of these terms are not very useful since they are specific to a paper or even a paragraph in a research paper. Hence, it is time to prune our vocabulary and start removing terms. Leveraging document frequency is a great way to achieve this. By now, you probably realize that the document frequency of a term is basically the total number of times that term occurs across all the documents in a corpus.

```
# Filter out words that occur less than 20 documents, or more than 50% of
the documents.
dictionary.filter_extremes(no_below=20, no_above=0.6)
print('Total Vocabulary Size:', len(dictionary))
```

```
Total Vocabulary Size: 7756
```

We removed all terms that occur fewer than 20 times across all documents and all terms that occur in more than 60% of all the documents. We are interested in finding different themes and topics and not recurring themes. Hence, this suits our scenario perfectly. We can now perform feature engineering by leveraging a simple Bag of Words model.

```
# Transforming corpus into bag of words vectors
bow_corpus = [dictionary.doc2bow(text) for text in norm_corpus_bigrams]
print(bow_corpus[1][:50])
```

```
[(4, 1), (14, 2), (20, 1), (28, 1), (33, 1), (43, 1), (50, 1), (60, 2),
(61, 1), (62, 2), (63, 1), (72, 1), (84, 1), ..., (286, 39), (296, 6),
(306, 1), (307, 2), (316, 1)]
```

```
# viewing actual terms and their counts
print([(dictionary[idx] , freq) for idx, freq in bow_corpus[1][:50]])
```

```
[('achieved', 1), ('allow', 2), ('american_institute', 1), ('another', 1),
('appeared', 1), ('argument', 1), ('assume', 1), ('become', 2),
('becomes', 1), ('behavior', 2), ('behavioral', 1), ('bounded', 1),
('cause', 1), ..., ('group', 39), ('hence', 6), ('implementation', 1),
('implemented', 2), ('independent', 1)]
```

```
# total papers in the corpus
print('Total number of papers:', len(bow_corpus))
```

```
Total number of papers: 1740
```

Our documents are now processed and have a good enough representation with the Bag of Words model to begin modeling.

Latent Semantic Indexing

Our first technique is Latent Semantic Indexing (LSI), which was developed in the 1970s as a statistical technique to correlate semantically linked terms from corpora. LSI is not just used for text summarization, but also in information retrieval and search. LSI uses the very popular Singular Value Decomposition (SVD) technique, which we discussed in detail in the "Important Concepts" section. The main principle behind LSI is that similar terms tend to be used in the same context and hence tend to co-occur more. The term LSI comes from the fact that this technique has the ability to uncover latent hidden terms that correlate semantically to form topics.

We now implement an LSI by leveraging Gensim and extract topics from our corpus on NIPS research papers. It is quite simple to build this model, thanks to Gensim's clean and concise API.

```
%%time

TOTAL_TOPICS = 10
lsi_bow = gensim.models.LsiModel(bow_corpus, id2word=dictionary,
num_topics=TOTAL_TOPICS, onepass=True, chunksize=1740, power_iters=1000)
```

```
CPU times: user 54min 30s, sys: 3min 21s, total: 57min 51s
Wall time: 3min 51s
```

Once the model is built, we can view the major topics or themes in our corpus by using the following code. Remember we had explicitly set the number of topics to 10 in this case.

```
for topic_id, topic in lsi_bow.print_topics(num_topics=10, num_words=20):
    print('Topic #'+str(topic_id+1)+':')
    print(topic)
    print()
```

```
Topic #1:
0.215*"unit" + 0.212*"state" + 0.187*"training" + 0.177*"neuron" +
0.162*"pattern" + 0.145*"image" + 0.140*"vector" + 0.125*"feature"
+ 0.122*"cell" + 0.110*"layer" + 0.101*"task" + 0.097*"class" +
0.091*"probability" + 0.089*"signal" + 0.087*"step" + 0.086*"response"
+ 0.085*"representation" + 0.083*"noise" + 0.082*"rule" +
0.081*"distribution"
```

```
Topic #2:
-0.487*"neuron" + -0.396*"cell" + 0.257*"state" + -0.191*"response" +
0.187*"training" + -0.170*"stimulus" + -0.117*"activity" + 0.109*"class" +
-0.099*"spike" + -0.097*"pattern" + -0.096*"circuit" + -0.096*"synaptic"
+ 0.095*"vector" + -0.090*"signal" + -0.090*"firing" + -0.088*"visual" +
0.084*"classifier" + 0.083*"action" + 0.078*"word" + -0.078*"cortical"

Topic #3:
-0.627*"state" + 0.395*"image" + -0.219*"neuron" + 0.209*"feature" +
-0.188*"action" + 0.137*"unit" + 0.131*"object" + -0.130*"control" +
0.129*"training" + -0.109*"policy" + 0.103*"classifier" + 0.090*"class"
+ -0.081*"step" + -0.081*"dynamic" + 0.080*"classification" +
0.078*"layer" + 0.076*"recognition" + -0.074*"reinforcement_learning" +
0.069*"representation" + 0.068*"pattern"

Topic #4:
-0.686*"unit" + 0.433*"image" + -0.182*"pattern" + -0.131*"layer" +
-0.123*"hidden_unit" + -0.121*"net" + -0.114*"training" + 0.112*"feature"
+ -0.109*"activation" + -0.107*"rule" + 0.097*"neuron" + -0.078*"word"
+ 0.070*"pixel" + -0.070*"connection" + 0.067*"object" + 0.065*"state"
+ 0.060*"distribution" + 0.059*"face" + -0.057*"architecture" +
0.055*"estimate"

Topic #5:
-0.428*"image" + -0.348*"state" + 0.266*"neuron" + -0.264*"unit" +
0.181*"training" + 0.174*"class" + -0.168*"object" + 0.167*"classifier"
+ -0.147*"action" + -0.122*"visual" + 0.117*"vector" + 0.115*"node"
+ 0.105*"distribution" + -0.103*"motion" + -0.099*"feature" +
0.097*"classification" + -0.097*"control" + -0.095*"task" + -0.087*"cell" +
-0.083*"representation"

Topic #6:
0.660*"cell" + -0.508*"neuron" + -0.213*"image" + -0.103*"chip" +
-0.097*"unit" + 0.093*"response" + -0.090*"object" + 0.083*"rat"
+ 0.076*"distribution" + -0.070*"circuit" + 0.069*"probability" +
0.064*"stimulus" + -0.061*"memory" + -0.058*"analog" + -0.058*"activation"
+ 0.055*"class" + -0.053*"bit" + -0.052*"net" + 0.051*"cortical" +
0.050*"firing"
```

```
Topic #7:
-0.353*"word" + 0.281*"unit" + -0.272*"training" + -0.257*"classifier"
+ -0.177*"recognition" + 0.159*"distribution" + -0.152*"feature" +
-0.144*"state" + -0.142*"pattern" + 0.141*"vector" + -0.128*"cell" +
-0.128*"task" + 0.122*"approximation" + 0.121*"variable" + 0.110*"equation"
+ -0.107*"classification" + 0.106*"noise" + -0.103*"class" + 0.101*"matrix"
+ -0.098*"neuron"

Topic #8:
-0.303*"pattern" + 0.243*"signal" + 0.236*"control" + 0.202*"training"
+ -0.181*"rule" + -0.178*"state" + 0.167*"noise" + -0.166*"class"
+ 0.162*"word" + -0.155*"cell" + -0.154*"feature" + 0.147*"motion"
+ 0.140*"task" + -0.127*"node" + -0.124*"neuron" + 0.116*"target"
+ 0.114*"circuit" + -0.114*"probability" + -0.110*"classifier" +
-0.109*"image"

Topic #9:
-0.472*"node" + -0.254*"circuit" + 0.214*"word" + -0.201*"chip" +
0.190*"neuron" + 0.172*"stimulus" + -0.160*"classifier" + -0.152*"current" +
0.147*"feature" + -0.146*"voltage" + 0.145*"distribution" + -0.141*"control"
+ -0.124*"rule" + -0.110*"layer" + -0.105*"analog" + -0.091*"tree" +
0.084*"response" + 0.080*"state" + 0.079*"probability" + 0.079*"estimate"

Topic #10:
0.518*"word" + -0.254*"training" + 0.236*"vector" + -0.222*"task" +
-0.194*"pattern" + -0.156*"classifier" + 0.149*"node" + 0.146*"recognition"
+ -0.139*"control" + 0.138*"sequence" + -0.126*"rule" + 0.125*"circuit"
+ 0.123*"cell" + -0.113*"action" + -0.105*"neuron" + 0.094*"hmm" +
0.093*"character" + 0.088*"chip" + 0.088*"matrix" + 0.085*"structure"
```

Let's take a moment to understand these results. A brief recap on the LSI model—it is based on the principle that words that are used in the same contexts tend to have similar meanings. You can observe in this output that each topic is a combination of terms (which basically tend to convey an overall sense of the topic) and weights. Now the problem here is that we have both positive and negative weights. What does that mean?

Based on existing research and my interpretations, considering we are reducing the dimensionality here to a 10-dimensional space based on the number of topics, the sign on each term indicates a sense of direction or orientation in the vector space for a particular topic. The higher the weight, the more important the contribution. So similar correlated terms have the same sign or direction. Hence, it is perfectly possible for a topic to have two different sub-themes based on the sign or orientation of terms. Let's separate these terms and try to interpret the topics again.

```
for n in range(TOTAL_TOPICS):
    print('Topic #'+str(n+1)+':')
    print('='*50)
    d1 = []
    d2 = []
    for term, wt in lsi_bow.show_topic(n, topn=20):
        if wt >= 0:
            d1.append((term, round(wt, 3)))
        else:
            d2.append((term, round(wt, 3)))

    print('Direction 1:', d1)
    print('-'*50)
    print('Direction 2:', d2)
    print('-'*50)
    print()
```

```
Topic #1:
==================================================
Direction 1: [('unit', 0.215), ('state', 0.212), ('training', 0.187),
('neuron', 0.177), ('pattern', 0.162), ('image', 0.145), ('vector', 0.14),
('feature', 0.125), ('cell', 0.122), ('layer', 0.11), ('task', 0.101),
('class', 0.097), ('probability', 0.091), ('signal', 0.089), ('step',
0.087), ('response', 0.086), ('representation', 0.085), ('noise', 0.083),
('rule', 0.082), ('distribution', 0.081)]
--------------------------------------------------
Direction 2: []
--------------------------------------------------
```

```
Topic #2:
==================================================
Direction 1: [('state', 0.257), ('training', 0.187), ('class', 0.109),
('vector', 0.095), ('classifier', 0.084), ('action', 0.083), ('word',
0.078)]
--------------------------------------------------
Direction 2: [('neuron', -0.487), ('cell', -0.396), ('response', -0.191),
('stimulus', -0.17), ('activity', -0.117), ('spike', -0.099), ('pattern',
-0.097), ('circuit', -0.096), ('synaptic', -0.096), ('signal', -0.09),
('firing', -0.09), ('visual', -0.088), ('cortical', -0.078)]
--------------------------------------------------

Topic #3:
==================================================
Direction 1: [('image', 0.395), ('feature', 0.209), ('unit', 0.137),
('object', 0.131), ('training', 0.129), ('classifier', 0.103), ('class',
0.09), ('classification', 0.08), ('layer', 0.078), ('recognition', 0.076),
('representation', 0.069), ('pattern', 0.068)]
--------------------------------------------------
Direction 2: [('state', -0.627), ('neuron', -0.219), ('action', -0.188),
('control', -0.13), ('policy', -0.109), ('step', -0.081), ('dynamic',
-0.081), ('reinforcement_learning', -0.074)]
--------------------------------------------------

Topic #4:
==================================================
Direction 1: [('image', 0.433), ('feature', 0.112), ('neuron', 0.097),
('pixel', 0.07), ('object', 0.067), ('state', 0.065), ('distribution',
0.06), ('face', 0.059), ('estimate', 0.055)]
--------------------------------------------------
Direction 2: [('unit', -0.686), ('pattern', -0.182), ('layer', -0.131),
('hidden_unit', -0.123), ('net', -0.121), ('training', -0.114),
('activation', -0.109), ('rule', -0.107), ('word', -0.078), ('connection',
-0.07), ('architecture', -0.057)]
--------------------------------------------------
```

```
Topic #5:
==================================================
Direction 1: [('neuron', 0.266), ('training', 0.181), ('class', 0.174),
('classifier', 0.167), ('vector', 0.117), ('node', 0.115), ('distribution',
0.105), ('classification', 0.097)]
--------------------------------------------------
Direction 2: [('image', -0.428), ('state', -0.348), ('unit', -0.264),
('object', -0.168), ('action', -0.147), ('visual', -0.122), ('motion',
-0.103), ('feature', -0.099), ('control', -0.097), ('task', -0.095),
('cell', -0.087), ('representation', -0.083)]

--------------------------------------------------

Topic #6:
==================================================
Direction 1: [('cell', 0.66), ('response', 0.093), ('rat', 0.083),
('distribution', 0.076), ('probability', 0.069), ('stimulus', 0.064),
('class', 0.055), ('cortical', 0.051), ('firing', 0.05)]
--------------------------------------------------
Direction 2: [('neuron', -0.508), ('image', -0.213), ('chip', -0.103),
('unit', -0.097), ('object', -0.09), ('circuit', -0.07), ('memory',
-0.061), ('analog', -0.058), ('activation', -0.058), ('bit', -0.053),
('net', -0.052)]
--------------------------------------------------

Topic #7:
==================================================
Direction 1: [('unit', 0.281), ('distribution', 0.159), ('vector', 0.141),
('approximation', 0.122), ('variable', 0.121), ('equation', 0.11),
('noise', 0.106), ('matrix', 0.101)]
--------------------------------------------------
Direction 2: [('word', -0.353), ('training', -0.272), ('classifier',
-0.257), ('recognition', -0.177), ('feature', -0.152), ('state', -0.144),
('pattern', -0.142), ('cell', -0.128), ('task', -0.128), ('classification',
-0.107), ('class', -0.103), ('neuron', -0.098)]
--------------------------------------------------
```

```
Topic #8:
==================================================
Direction 1: [('signal', 0.243), ('control', 0.236), ('training', 0.202),
('noise', 0.167), ('word', 0.162), ('motion', 0.147), ('task', 0.14),
('target', 0.116), ('circuit', 0.114)]
--------------------------------------------------
Direction 2: [('pattern', -0.303), ('rule', -0.181), ('state', -0.178),
('class', -0.166), ('cell', -0.155), ('feature', -0.154), ('node', -0.127),
('neuron', -0.124), ('probability', -0.114), ('classifier', -0.11),
('image', -0.109)]
--------------------------------------------------

Topic #9:
==================================================
Direction 1: [('word', 0.214), ('neuron', 0.19), ('stimulus', 0.172),
('feature', 0.147), ('distribution', 0.145), ('response', 0.084), ('state',
0.08), ('probability', 0.079), ('estimate', 0.079)]
--------------------------------------------------
Direction 2: [('node', -0.472), ('circuit', -0.254), ('chip', -0.201),
('classifier', -0.16), ('current', -0.152), ('voltage', -0.146),
('control', -0.141), ('rule', -0.124), ('layer', -0.11), ('analog',
-0.105), ('tree', -0.091)]
--------------------------------------------------

Topic #10:
==================================================
Direction 1: [('word', 0.518), ('vector', 0.236), ('node', 0.149),
('recognition', 0.146), ('sequence', 0.138), ('circuit', 0.125), ('cell',
0.123), ('hmm', 0.094), ('character', 0.093), ('chip', 0.088), ('matrix',
0.088), ('structure', 0.085)]
--------------------------------------------------
Direction 2: [('training', -0.254), ('task', -0.222), ('pattern', -0.194),
('classifier', -0.156), ('control', -0.139), ('rule', -0.126), ('action',
-0.113), ('neuron', -0.105)]
--------------------------------------------------
```

Does this make things better? Well, it's definitely a lot better than the previous interpretation. Here we can see clear themes of modeling being applied in chips and electronic devices, classification and recognition models, neural models talking about the human brain components like cells, stimuli, neurons, cortical components, and even themes around reinforcement learning! We explore these in detail later in a more structured way.

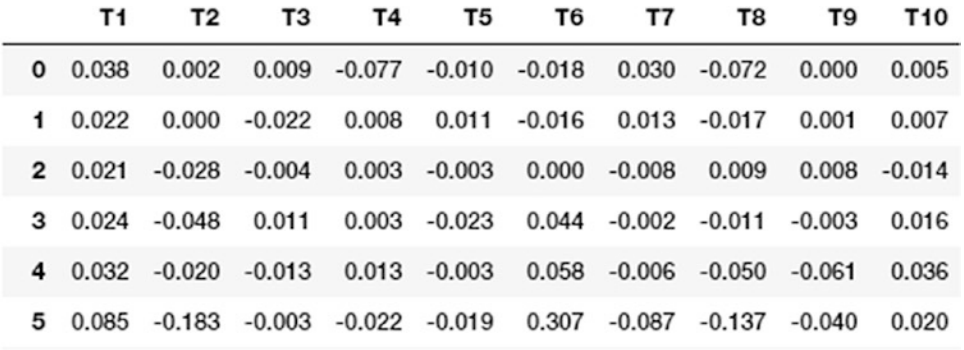
Let's try to get the three major matrices (**U**, **S**, and **VT**) from our topic model, which uses SVD (based on the foundational concepts mentioned earlier).

```
term_topic = lsi_bow.projection.u
singular_values = lsi_bow.projection.s
topic_document = (gensim.matutils.corpus2dense(lsi_bow[bow_corpus],
len(singular_values)).T / singular_values).T
term_topic.shape, singular_values.shape, topic_document.shape
```

```
((7756, 10), (10,), (10, 1740))
```

Just like the preceding output shows, we have a term-topic matrix, singular values, and a topic-document matrix. We can transpose the topic-document matrix to form a document-topic matrix and that would help us see the proportion of each topic per document (a larger proportion means the topic is more dominant in the document). See Figure 6-2.

```
document_topics = pd.DataFrame(np.round(topic_document.T, 3),
                               columns=['T'+str(i) for i in range(1, TOTAL_
                               TOPICS+1)])
document_topics.head(5)
```



	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
0	0.038	0.002	0.009	-0.077	-0.010	-0.018	0.030	-0.072	0.000	0.005
1	0.022	0.000	-0.022	0.008	0.011	-0.016	0.013	-0.017	0.001	0.007
2	0.021	-0.028	-0.004	0.003	-0.003	0.000	-0.008	0.009	0.008	-0.014
3	0.024	-0.048	0.011	0.003	-0.023	0.044	-0.002	-0.011	-0.003	0.016
4	0.032	-0.020	-0.013	0.013	-0.003	0.058	-0.006	-0.050	-0.061	0.036
5	0.085	-0.183	-0.003	-0.022	-0.019	0.307	-0.087	-0.137	-0.040	0.020

***Figure 6-2.** Document-topic matrix from our LSI model*

Ignoring the sign, we can try to find out the most important topics for a few sample papers and see if they make sense.

```
document_numbers = [13, 250, 500]

for document_number in document_numbers:
    top_topics = list(document_topics.columns[np.argsort(-
                                          np.absolute(
                                      document_topics.iloc[document_
                                      number].values))[:3]])
    print('Document #'+str(document_number)+':')
    print('Dominant Topics (top 3):', top_topics)
    print('Paper Summary:')
    print(papers[document_number][:500])
    print()
```

```
Document #13:
Dominant Topics (top 3): ['T6', 'T1', 'T2']
Paper Summary:
9
Stochastic Learning Networks and their Electronic Implementation
Joshua Alspector*, Robert B. Allen, Victor Hut, and Srinagesh Satyanarayana
Bell Communications Research, Morristown, NJ 07960
ABSTRACT
We describe a family of learning algorithms that operate on a recurrent,
symmetrically connected, neuromorphic network that, like the Boltzmann
machine, settles in the presence of noise. These networks learn by
modifying synaptic connection strengths on the basis of correlations
seen loca

Document #250:
Dominant Topics (top 3): ['T3', 'T5', 'T8']
Paper Summary:
266 Zemel, Mozer and Hinton
TRAFFIC: Recognizing Objects Using
Hierarchical Reference Frame Transformations
```

```
Richard S. Zemel
Computer Science Dept.
University of Toronto
...
ABSTRACT
We describe a model that can recognize two-dimensional shapes in
an unsegmented image, independent of their orie

Document #500:
Dominant Topics (top 3): ['T9', 'T8', 'T10']
Paper Summary:
Constrained Optimization Applied to the
Parameter Setting Problem for Analog Circuits
David Kirk, Kurt Fleischer, Lloyd Watts, Alan Bart
Computer Graphics 350-74
California Institute of Technology
Abstract
We use constrained optimization to select operating parameters for two
circuits: a simple 3-transistor square root circuit, and an analog VLSI
artificial cochlea. This automated method uses computer controlled mea-
surement and test equipment to choose chip paramet
```

If you look at the description of the terms in each of the selected topics in the preceding output, they make perfect sense.

- Paper #13 has a dominance of topics 6, 1, and 2, which pretty much talk about neurons, cells, brain's cortex, stimulus, and so on (aspects around neuromorphic networks).
- Paper #250 has a dominance of topics 3, 5, and 8, which talk about object recognition, image classification, and visual representation with neural networks. This matches the paper's theme, which is about object recognition.
- Paper #500 has a dominance of topics 9, 8, and 10, which talk about signals, voltage, chips, circuits, and so on. This is in line with the theme of the paper around parameter settings for analog circuits.

This shows us that the LSI model is quite effective, although a tad difficult to interpret based on the positive and negative weights, which often make things confusing.

Implementing LSI Topic Models from Scratch

Based on what we mentioned earlier, the heart of LSI models involves Singular Value Decomposition (SVD). Here, we try to implement an LSI topic model from scratch using low-rank SVD. The first step in SVD is to get the source matrix, which is typically a term-document matrix. We can obtain it from Gensim by converting the sparse Bag of Words representation into a dense matrix.

```
td_matrix = gensim.matutils.corpus2dense(corpus=bow_corpus,
num_terms=len(dictionary))
print(td_matrix.shape)
td_matrix
```

```
(7756, 1740)
array([[1., 0., 1., ..., 0., 2., 1.],
       [1., 0., 1., ..., 1., 1., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Everything seems to be in order, so we can validate our vocabulary by using the following code just to make sure everything is correct.

```
vocabulary = np.array(list(dictionary.values()))
print('Total vocabulary size:', len(vocabulary))
vocabulary
```

```
Total vocabulary size: 7756
array(['able', 'abstract', 'accommodate', ..., 'support_vector',
       'mozer_jordan', 'kearns_solla'], dtype='<U28')
```

We now perform low-rank SVD on our term document matrix by leveraging the following code snippet.

```
from scipy.sparse.linalg import svds

u, s, vt = svds(td_matrix, k=TOTAL_TOPICS, maxiter=10000)
term_topic = u
```

```
singular_values = s
topic_document = vt
term_topic.shape, singular_values.shape, topic_document.shape
```

```
((7756, 10), (10,), (10, 1740))
```

Getting the weights (direction and importance) for each term in each topic is also pretty straightforward. The following code helps us compute this.

```
tt_weights = term_topic.transpose() * singular_values[:, None]
tt_weights.shape
```

```
(10, 7756)
```

We can now easily look at our 10 topics and the top influential terms for them by using the following code.

```
top_terms = 20
topic_key_term_idxs = np.argsort(-np.absolute(tt_weights), axis=1)[:, :top_
terms]
topic_keyterm_weights = np.array([tt_weights[row, columns]
                             for row, columns in list(zip(np.arange(TOTAL_
                             TOPICS), topic_key_term_idxs))])
topic_keyterms = vocabulary[topic_key_term_idxs]
topic_keyterms_weights = list(zip(topic_keyterms, topic_keyterm_weights))
for n in range(TOTAL_TOPICS):
    print('Topic #'+str(n+1)+':')
    print('='*50)
    d1 = []
    d2 = []
    terms, weights = topic_keyterms_weights[n]
    term_weights = sorted([(t, w) for t, w in zip(terms, weights)],
                          key=lambda row: -abs(row[1]))
    for term, wt in term_weights:
        if wt >= 0:
            d1.append((term, round(wt, 3)))
        else:
            d2.append((term, round(wt, 3)))
    print('Direction 1:', d1)
    print('-'*50)
    print('Direction 2:', d2)
    print('-'*50)
    print()
```

```
Topic #1:
==================================================
Direction 1: [('training', 92.618), ('task', 80.732), ('pattern', 70.619),
('classifier', 56.989), ('control', 50.677), ('rule', 45.926), ('action',
41.202), ('neuron', 38.193)]
--------------------------------------------------
Direction 2: [('word', -188.488), ('vector', -85.973), ('node', -54.376),
('recognition', -53.232), ('sequence', -50.351), ('circuit', -45.394),
('cell', -44.811), ('hmm', -34.086), ('character', -34.022), ('chip',
-32.16), ('matrix', -32.093), ('structure', -30.993)]
--------------------------------------------------

Topic #2:
==================================================
Direction 1: [('word', 78.347), ('neuron', 69.793), ('stimulus', 63.234),
('feature', 53.819), ('distribution', 53.119), ('response', 30.954),
('state', 29.343), ('probability', 29.099), ('estimate', 28.908)]
--------------------------------------------------
Direction 2: [('node', -173.277), ('circuit', -93.0), ('chip', -73.593),
('classifier', -58.717), ('current', -55.844), ('voltage', -53.489),
('control', -51.708), ('rule', -45.293), ('layer', -40.265), ('analog',
-38.344), ('tree', -33.483)]
--------------------------------------------------

Topic #3:
==================================================
Direction 1: [('pattern', 116.971), ('rule', 69.783), ('state', 68.605),
('class', 64.259), ('cell', 59.979), ('feature', 59.606), ('node', 49.175),
('neuron', 47.998), ('probability', 43.812), ('classifier', 42.612),
('image', 42.061)]
--------------------------------------------------
```

```
Direction 2: [('signal', -93.805), ('control', -91.041), ('training',
-77.88), ('noise', -64.397), ('word', -62.392), ('motion', -56.699),
('task', -53.883), ('target', -44.765), ('circuit', -44.129)]
--------------------------------------------------

Topic #4:
==================================================
Direction 1: [('unit', 117.727), ('distribution', 66.719), ('vector',
58.881), ('approximation', 50.931), ('variable', 50.83), ('equation',
46.229), ('noise', 44.247), ('matrix', 42.214)]
--------------------------------------------------
Direction 2: [('word', -147.792), ('training', -113.693), ('classifier',
-107.386), ('recognition', -73.948), ('feature', -63.454), ('state',
-60.126), ('pattern', -59.562), ('cell', -53.768), ('task', -53.693),
('classification', -44.936), ('class', -43.161), ('neuron', -41.092)]
--------------------------------------------------

Topic #5:
==================================================
Direction 1: [('neuron', 220.116), ('image', 92.39), ('chip', 44.422),
('unit', 41.922), ('object', 39.001), ('circuit', 30.444), ('memory',
26.475), ('analog', 25.207), ('activation', 24.953), ('bit', 22.997),
('net', 22.699)]
--------------------------------------------------
Direction 2: [('cell', -285.803), ('response', -40.216), ('rat', -35.975),
('distribution', -33.085), ('probability', -29.79), ('stimulus', -27.789),
('class', -24.02), ('cortical', -22.185), ('firing', -21.66)]
--------------------------------------------------

Topic #6:
==================================================
Direction 1: [('image', 209.793), ('state', 170.207), ('unit', 129.108),
('object', 82.185), ('action', 72.136), ('visual', 59.502), ('motion',
50.605), ('feature', 48.665), ('control', 47.427), ('task', 46.496),
('cell', 42.366), ('representation', 40.564)]
--------------------------------------------------
```

```
Direction 2: [('neuron', -130.053), ('training', -88.668), ('class',
-85.213), ('classifier', -81.921), ('vector', -57.532), ('node', -56.341),
('distribution', -51.622), ('classification', -47.645)]
--------------------------------------------------

Topic #7:
==================================================
Direction 1: [('image', 215.858), ('feature', 55.647), ('neuron', 48.494),
('pixel', 35.095), ('object', 33.585), ('state', 32.544), ('distribution',
29.977), ('face', 29.256), ('estimate', 27.555)]
--------------------------------------------------
Direction 2: [('unit', -341.829), ('pattern', -90.771), ('layer',
-65.337), ('hidden_unit', -61.12), ('net', -60.035), ('training',
-56.742), ('activation', -54.268), ('rule', -53.377), ('word', -38.903),
('connection', -34.618), ('architecture', -28.439)]
--------------------------------------------------

Topic #8:
==================================================
Direction 1: [('image', 229.287), ('feature', 121.397), ('unit', 79.44),
('object', 76.204), ('training', 75.152), ('classifier', 59.872), ('class',
52.527), ('classification', 46.696), ('layer', 45.149), ('recognition',
44.192), ('representation', 40.179), ('pattern', 39.252)]
--------------------------------------------------
Direction 2: [('state', -364.388), ('neuron', -127.022), ('action',
-109.245), ('control', -75.369), ('policy', -63.103), ('step', -47.226),
('dynamic', -46.907), ('reinforcement_learning', -42.747)]
--------------------------------------------------

Topic #9:
==================================================
Direction 1: [('neuron', 306.151), ('cell', 249.243), ('response',
119.758), ('stimulus', 106.762), ('activity', 73.499), ('spike', 62.039),
('pattern', 60.957), ('circuit', 60.602), ('synaptic', 60.282), ('signal',
56.665), ('firing', 56.597), ('visual', 55.571), ('cortical', 48.867)]
--------------------------------------------------
```

```
Direction 2: [('state', -161.465), ('training', -117.32), ('class',
-68.732), ('vector', -59.558), ('classifier', -52.589), ('action',
-52.113), ('word', -49.239)]
--------------------------------------------------

Topic #10:
==================================================
Direction 1: []
--------------------------------------------------
Direction 2: [('unit', -260.793), ('state', -258.146), ('training',
-227.312), ('neuron', -215.681), ('pattern', -197.232), ('image',
-175.735), ('vector', -170.154), ('feature', -151.547), ('cell',
-148.138), ('layer', -133.593), ('task', -122.389), ('class', -117.849),
('probability', -110.526), ('signal', -108.232), ('step', -105.202),
('response', -104.465), ('representation', -103.255), ('noise', -100.573),
('rule', -99.611), ('distribution', -98.973)]
--------------------------------------------------
```

Note that even if the topic numbers have shuffled around, they match the previous topic model's output from Gensim very closely! You can also try extracting influential topics from sample papers using the following code.

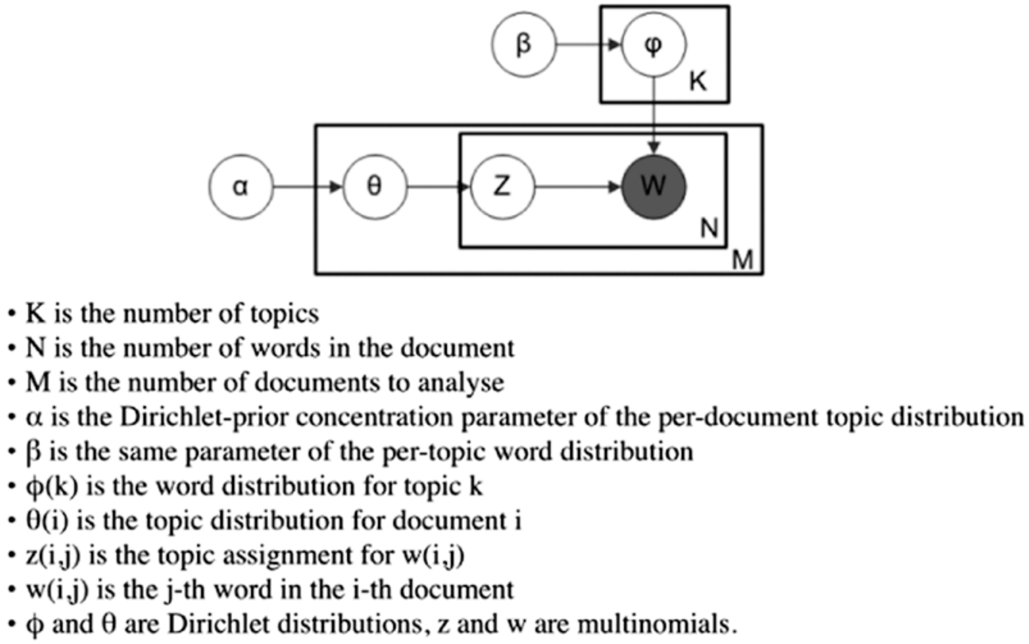
```
document_topics = pd.DataFrame(np.round(topic_document.T, 3),
                               columns=['T'+str(i) for i in range(1, TOTAL_
                               TOPICS+1)])
document_numbers = [13, 250, 500]

for document_number in document_numbers:
    top_topics = list(document_topics.columns[np.argsort(-
                                                    np.absolute(
                                            document_topics.iloc[document_
                                            number].values))[:3]])
    print('Document #'+str(document_number)+':')
    print('Dominant Topics (top 3):', top_topics)
    print('Paper Summary:')
    print(papers[document_number][:500])
    print()
```

```
Document #13:
Dominant Topics (top 3): ['T5', 'T10', 'T9']
Paper Summary:
9
Stochastic Learning Networks and their Electronic Implementation
Joshua Alspector*, Robert B. Allen, Victor Hut, and Srinagesh Satyanarayana
Bell Communications Research, Morristown, NJ 07960
ABSTRACT
We describe a family of learning algorithms that operate on a recurrent,
symmetrically connected, neuromorphic network that, like the Boltzmann machine

Document #250:
Dominant Topics (top 3): ['T6', 'T8', 'T3']
Paper Summary:
266 Zemel, Mozer and Hinton
TRAFFIC: Recognizing Objects Using
Hierarchical Reference Frame Transformations
Richard S. Zemel
Computer Science Dept.
University of Toronto
...
ABSTRACT
We describe a model that can recognize two-dimensional shapes in
an unsegmented image, independent of their orie

Document #500:
Dominant Topics (top 3): ['T2', 'T3', 'T1']
Paper Summary:
Constrained Optimization Applied to the
Parameter Setting Problem for Analog Circuits
David Kirk, Kurt Fleischer, Lloyd Watts, Alan Bart
...
Abstract
We use constrained optimization to select operating parameters for two
circuits: a simple 3-transistor square root circuit, and an analog VLSI
artificial cochlea.
```

We can clearly observe that even if the topic numbers have changed (since they were shuffled around based on our new model), the core themes pertaining to each topic closely match the results we obtained earlier. Clearly SVD is a very powerful mathematical operation and we will see more of this during document summarization!

Latent Dirichlet Allocation

The Latent Dirichlet Allocation (LDA) technique is a generative probabilistic model in which each document is assumed to have a combination of topics similar to a probabilistic Latent Semantic Indexing model. In this case, the latent topics contain a Dirichlet prior over them. The math behind in this technique is pretty involved, so we will try to summarize it since going it specific details is out of the current scope. We recommend readers to go through this excellent talk at http://chdoig.github.io/pygotham-topic-modeling/#/ by Christine Doig from which we will be borrowing some excellent pictorial representations. The plate notation for the LDA model is depicted in Figure 6-3.



- K is the number of topics
- N is the number of words in the document
- M is the number of documents to analyse
- α is the Dirichlet-prior concentration parameter of the per-document topic distribution
- β is the same parameter of the per-topic word distribution
- $\phi(k)$ is the word distribution for topic k
- $\theta(i)$ is the topic distribution for document i
- $z(i,j)$ is the topic assignment for $w(i,j)$
- $w(i,j)$ is the j-th word in the i-th document
- ϕ and θ are Dirichlet distributions, z and w are multinomials.

Figure 6-3. *LDA plate notation (courtesy C. Doug. Introduction to Topic Modeling in Python)*

You can find more details about the diagram in Figure 6-3 in the official Wikipedia article at https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation, which talks about the parameters in detail. Figure 6-3 gives us a good representation of how each of the parameters connects to the text documents and terms. It is assumed that we have **M** documents, **N** number of words in the documents, and **K** total number of topics to generate.

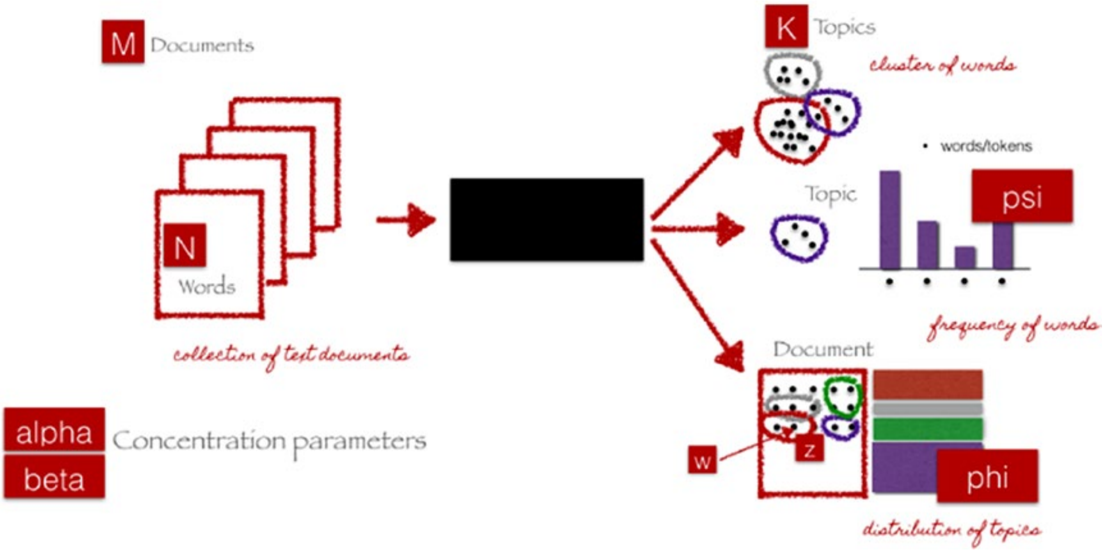


Figure 6-4. *End-to-end LDA framework (courtesy C. Doug. Introduction to Topic Modeling in Python)*

The black box in Figure 6-4 represents the core algorithm, which uses the previously mentioned parameters to extract **K** topics from the documents. The following steps give a very simplistic explanation of what happens in the algorithm for everyone's benefit.

1. Initialize the necessary parameters.
2. For each document, randomly initialize each word to one of the **K** topics.
3. Start an iterative process as follows and repeat it several times. For each document **D,** for each word ***W*** in document, and for each topic **T:**
    - Compute $P(T|D)$, which is proportion of words in **D** assigned to topic **T**.
    - Compute $P(W|T)$, which is proportion of assignments to topic **T** over all documents having the word **W**.
    - Reassign word **W** with topic **T** with probability $P(T|D) \times P(W|T)$, considering all other words and their topic assignments.

Once this runs several iterations, we should have topic mixtures for each document and then we can generate the constituents of each topic from the terms that point to that topic. Popularly, the method used here is known as *Collapsed Gibbs Sampling*. We use Gensim in the following implementation to build an LDA-based topic model on our research paper based corpus.

```
%%time
lda_model = gensim.models.LdaModel(corpus=bow_corpus, id2word=dictionary,
                                   chunksize=1740, alpha='auto',
                                   eta='auto', random_state=42,
                                   iterations=500, num_topics=TOTAL_TOPICS,
                                   passes=20, eval_every=None)
```

```
CPU times: user 5min 32s, sys: 10.7 s, total: 5min 43s
Wall time: 2min 31s
```

Viewing the topics in our trained topic model is quite easy and we can generate them with the following code.

```
for topic_id, topic in lda_model.print_topics(num_topics=10, num_words=20):
    print('Topic #'+str(topic_id+1)+':')
    print(topic)
    print()
```

```
Topic #1:
0.016*"training" + 0.012*"classifier" + 0.007*"pattern" +
0.007*"classification" + 0.006*"class" + 0.006*"task" + 0.006*"vector" +
0.005*"training_set" + 0.005*"feature" + 0.004*"control" + 0.004*"size"
+ 0.003*"trained" + 0.003*"teacher" + 0.003*"rate" + 0.003*"student"
+ 0.003*"average" + 0.003*"robot" + 0.003*"random" + 0.003*"rule" +
0.003*"search"

Topic #2:
0.008*"vector" + 0.006*"equation" + 0.006*"matrix" + 0.006*"neuron"
+ 0.005*"state" + 0.005*"dynamic" + 0.005*"solution" + 0.005*"unit"
+ 0.004*"node" + 0.004*"pattern" + 0.004*"linear" + 0.004*"let" +
0.003*"layer" + 0.003*"convergence" + 0.003*"rule" + 0.003*"size" +
0.003*"theorem" + 0.003*"threshold" + 0.003*"memory" + 0.003*"theory"
```

```
Topic #3:
0.017*"training" + 0.011*"word" + 0.008*"recognition" + 0.007*"trained"
+ 0.006*"net" + 0.006*"unit" + 0.006*"feature" + 0.006*"speech" +
0.006*"task" + 0.005*"architecture" + 0.005*"class" + 0.005*"character" +
0.004*"layer" + 0.004*"classification" + 0.004*"context" + 0.004*"test"
+ 0.004*"sequence" + 0.004*"hidden_unit" + 0.004*"experiment" +
0.004*"vector"

Topic #4:
0.017*"motion" + 0.009*"rule" + 0.008*"direction" + 0.007*"stimulus"
+ 0.006*"velocity" + 0.006*"task" + 0.006*"human" + 0.006*"unit" +
0.005*"location" + 0.005*"target" + 0.005*"subject" + 0.005*"memory" +
0.005*"prediction" + 0.005*"position" + 0.004*"concept" + 0.004*"field" +
0.004*"response" + 0.004*"cue" + 0.004*"layer" + 0.004*"hand"

Topic #5:
0.008*"distribution" + 0.005*"estimate" + 0.005*"sample" + 0.005*"training"
+ 0.005*"class" + 0.005*"probability" + 0.005*"approximation" +
0.004*"variable" + 0.004*"gaussian" + 0.004*"linear" + 0.004*"vector"
+ 0.004*"prior" + 0.004*"noise" + 0.004*"density" + 0.004*"prediction"
+ 0.003*"kernel" + 0.003*"variance" + 0.003*"mixture" + 0.003*"bound" +
0.003*"regression"

Topic #6:
0.037*"state" + 0.011*"action" + 0.008*"step" + 0.008*"control" +
0.007*"policy" + 0.006*"sequence" + 0.006*"reinforcement_learning" +
0.005*"probability" + 0.005*"optimal" + 0.004*"task" + 0.004*"transition"
+ 0.004*"environment" + 0.003*"variable" + 0.003*"reward" +
0.003*"stochastic" + 0.003*"goal" + 0.003*"machine" + 0.003*"current" +
0.003*"controller" + 0.003*"agent"

Topic #7:
0.012*"circuit" + 0.011*"signal" + 0.011*"chip" + 0.009*"neuron" +
0.008*"current" + 0.007*"voltage" + 0.006*"analog" + 0.006*"control"
+ 0.005*"channel" + 0.004*"noise" + 0.004*"neural" + 0.004*"bit" +
0.004*"implementation" + 0.004*"source" + 0.003*"design" + 0.003*"gain" +
0.003*"processor" + 0.003*"synapse" + 0.003*"device" + 0.003*"array"
```

```
Topic #8:
0.021*"neuron" + 0.019*"cell" + 0.009*"response" + 0.007*"activity" +
0.007*"stimulus" + 0.007*"pattern" + 0.006*"spike" + 0.005*"synaptic" +
0.004*"cortical" + 0.004*"neural" + 0.004*"signal" + 0.004*"firing" +
0.004*"connection" + 0.004*"effect" + 0.004*"layer" + 0.004*"et_al" +
0.004*"cortex" + 0.003*"visual" + 0.003*"simulation" + 0.003*"synapsis"

Topic #9:
0.032*"image" + 0.012*"object" + 0.012*"feature" + 0.006*"pixel" +
0.006*"visual" + 0.005*"representation" + 0.005*"face" + 0.005*"vector" +
0.004*"view" + 0.004*"recognition" + 0.004*"transformation" + 0.004*"local"
+ 0.003*"map" + 0.003*"structure" + 0.003*"region" + 0.003*"filter" +
0.003*"position" + 0.003*"distance" + 0.003*"part" + 0.003*"location"

Topic #10:
0.030*"unit" + 0.009*"pattern" + 0.007*"representation" + 0.007*"activation"
+ 0.006*"hidden_unit" + 0.006*"node" + 0.006*"structure" + 0.006*"layer" +
0.005*"activity" + 0.004*"connection" + 0.004*"task" + 0.004*"component"
+ 0.004*"map" + 0.004*"rule" + 0.004*"architecture" + 0.004*"signal" +
0.004*"level" + 0.003*"response" + 0.003*"connectionist" + 0.003*"training"
```

The topics are definitely easier to understand and interpret than the LSI model, since all the weights are the same sign and tell us the importance of each term in the topic. We can also view the overall mean coherence score of the model.

```
topics_coherences = lda_model.top_topics(bow_corpus, topn=20)
avg_coherence_score = np.mean([item[1] for item in topics_coherences])
print('Avg. Coherence Score:', avg_coherence_score)
```

```
Avg. Coherence Score: -1.0433305600965899
```

Topic coherence is a complex topic in its own and it can be used to measure the quality of topic models to some extent. Typically, a set of statements is said to be coherent if they support each other. Topic models are unsupervised learning based models that are trained on unstructured text data, making it difficult to measure the quality of outputs. An excellent resource on the topic coherence framework is the paper by Michael Röder et al., “Exploring the Space of Topic Coherence Measures,” which you can access at http://svn.aksw.org/papers/2015/WSDM_Topic_Evaluation/public.pdf. The same framework has been implemented in Python in the Gensim framework.

A detailed article is available at `https://rare-technologies.com/what-is-topic-coherence` by Devashish who implemented this neat capability in Gensim!

I use his same easy-to-understand analogy to explain the topic coherence framework briefly where consider we have a water source and water is distributed to different people. To measure the water quality, we have to rely on individual customer reviews. Now assume we have four main hubs for water distribution and we install some equipment at the four water pipes distributing water in each of the hubs. This equipment helps us measure the quality of water using quantitative metrics, saving us time from relying on subjective reviews. Now take this analogy and consider the water is basically the topics we obtain from a topic model and the topic coherence framework is the equipment we installed to get a quantitative evaluation of the topic quality. This coherence framework is a four-stage pipeline, as defined in the research paper we mentioned earlier, and is depicted in Figure 6-5.

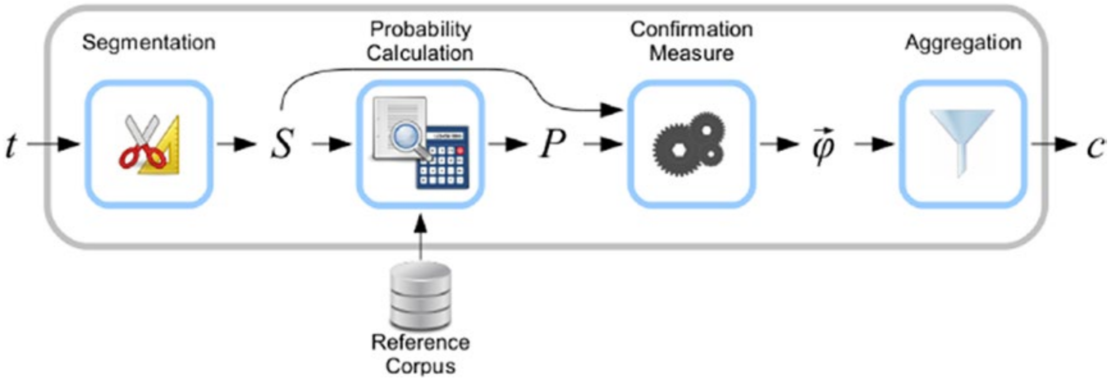


***Figure 6-5.** The unifying coherence framework for topic models*

The four main stages in the topic coherence framework pipeline depicted in Figure 6-5 are described as follows:

1. **Segmentation:** This stage is akin to when our water is partitioned into several glasses assuming that the quality of water in each glass is different. Here, the words in a topic are placed into subsets and pairs of words are created.

2. **Probability calculation:** The quantity of water in each glass is measured. The method of probability calculation or estimation defines the way that the probabilities are derived from the underlying data. The Boolean document estimates the probability of

a single word as the number of documents in which the word occurs by the total number of documents. Similarly, the joint probability of two words is estimated by the number of documents containing both words by the total number of documents. The Boolean sliding window determines word counts using a sliding window.

3. **Confirmation measure:** The quality of water (based on a certain metric) in each glass is measured and a number is assigned to each glass with regard to its quantity. Considering topic models, a confirmation measure takes a pair of words or word subsets as well as the corresponding probabilities and computes how strong the conditioning word set supports the other word in the pair. Typically there are two types of measures—extrinsic and intrinsic.

   - **Direct or extrinsic measures:** One of the most popular ones is the UCI measure. It uses the pointwise mutual information (PMI) as the pairwise scoring function. In the research paper, they came up with a direct metric C_p, which gave them the best performance (details mentioned in the paper).

   - **Indirect or intrinsic measures:** Indirect confirmation measures claim to capture semantic support that direct measures would miss. One of the most popular measures is the UMass measure, which uses the following scoring function.

     $$\mathrm{S}_{\mathrm{UMass}}\left(w_i,w_j\right)=\log\frac{D\left(w_i,w_j\right)+1}{D\left(w_i\right)}$$

     such that $D(w)$ is the document frequency of the term w. The scoring is basically the empirical conditional log-probability:

     $$\log p\left(w_j|w_i\right)=\log\frac{p\left(w_i,w_j\right)}{p\left(w_j\right)}$$

     which is smoothened by adding 1 to the document frequency in the numerator. The research paper claims to have found a new measure C_v, which is a combination of the indirect cosine measure and with the NPMI and the Boolean sliding window which gave the best results. This is also available in Gensim.

4. **Aggregation:** This is the equipment where these quality numbers are combined in a certain way (e.g., arithmetic mean) to come up with one quantitative metric. All confirmation measures for each subset word pairs per topic are aggregated to give a single coherence score. Just like we obtained -1.04 as the average coherence score in our previous output (the UMass measure).

This should give you enough context to evaluate and tune the topic models. Let's now look at the output of our LDA topic model in an easier to understand format. One way is to visualize the topics as tuples of terms and weights.

```
topics_with_wts = [item[0] for item in topics_coherences]
print('LDA Topics with Weights')
print('='*50)
for idx, topic in enumerate(topics_with_wts):
    print('Topic #'+str(idx+1)+':')
    print([(term, round(wt, 3)) for wt, term in topic])
    print()
```

```
LDA Topics with Weights
==================================================
Topic #1:
[('training', 0.017), ('word', 0.011), ('recognition', 0.008), ('trained',
0.007), ('net', 0.006), ('unit', 0.006), ('feature', 0.006), ('speech',
0.006), ('task', 0.006), ('architecture', 0.005), ('class', 0.005),
('character', 0.005), ('layer', 0.004), ('classification', 0.004),
('context', 0.004), ('test', 0.004), ('sequence', 0.004), ('hidden_unit',
0.004), ('experiment', 0.004), ('vector', 0.004)]

Topic #2:
[('unit', 0.03), ('pattern', 0.009), ('representation', 0.007),
('activation', 0.007), ('hidden_unit', 0.006), ('node', 0.006),
('structure', 0.006), ('layer', 0.006), ('activity', 0.005), ('connection',
0.004), ('task', 0.004), ('component', 0.004), ('map', 0.004), ('rule',
0.004), ('architecture', 0.004), ('signal', 0.004), ('level', 0.004),
('response', 0.003), ('connectionist', 0.003), ('training', 0.003)]
```

```
...,
...,

Topic #8:
[('state', 0.037), ('action', 0.011), ('step', 0.008), ('control', 0.008),
('policy', 0.007), ('sequence', 0.006), ('reinforcement_learning', 0.006),
('probability', 0.005), ('optimal', 0.005), ('task', 0.004), ('transition',
0.004), ('environment', 0.004), ('variable', 0.003), ('reward', 0.003),
('stochastic', 0.003), ('goal', 0.003), ('machine', 0.003), ('current',
0.003), ('controller', 0.003), ('agent', 0.003)]

Topic #9:
[('motion', 0.017), ('rule', 0.009), ('direction', 0.008), ('stimulus',
0.007), ('velocity', 0.006), ('task', 0.006), ('human', 0.006), ('unit',
0.006), ('location', 0.005), ('target', 0.005), ('subject', 0.005),
('memory', 0.005), ('prediction', 0.005), ('position', 0.005), ('concept',
0.004), ('field', 0.004), ('response', 0.004), ('cue', 0.004), ('layer',
0.004), ('hand', 0.004)]

Topic #10:
[('circuit', 0.012), ('signal', 0.011), ('chip', 0.011), ('neuron', 0.009),
('current', 0.008), ('voltage', 0.007), ('analog', 0.006), ('control',
0.006), ('channel', 0.005), ('noise', 0.004), ('neural', 0.004), ('bit',
0.004), ('implementation', 0.004), ('source', 0.004), ('design', 0.003),
('gain', 0.003), ('processor', 0.003), ('synapse', 0.003), ('device',
0.003), ('array', 0.003)]
```

We can also view the topics as a list of terms without the weights when we want to understand the context or theme conveyed by each topic.

```
print('LDA Topics without Weights')
print('='*50)
for idx, topic in enumerate(topics_with_wts):
    print('Topic #'+str(idx+1)+':')
    print([term for wt, term in topic])
    print()
```

```
LDA Topics without Weights
==================================================
Topic #1:
['training', 'word', 'recognition', 'trained', 'net', 'unit', 'feature',
'speech', 'task', 'architecture', 'class', 'character', 'layer',
'classification', 'context', 'test', 'sequence', 'hidden_unit',
'experiment', 'vector']

Topic #2:
['unit', 'pattern', 'representation', 'activation', 'hidden_unit', 'node',
'structure', 'layer', 'activity', 'connection', 'task', 'component', 'map',
'rule', 'architecture', 'signal', 'level', 'response', 'connectionist',
'training']

...,
...,

Topic #8:
['state', 'action', 'step', 'control', 'policy', 'sequence',
'reinforcement_learning', 'probability', 'optimal', 'task', 'transition',
'environment', 'variable', 'reward', 'stochastic', 'goal', 'machine',
'current', 'controller', 'agent']

Topic #9:
['motion', 'rule', 'direction', 'stimulus', 'velocity', 'task', 'human',
'unit', 'location', 'target', 'subject', 'memory', 'prediction',
'position', 'concept', 'field', 'response', 'cue', 'layer', 'hand']

Topic #10:
['circuit', 'signal', 'chip', 'neuron', 'current', 'voltage', 'analog',
'control', 'channel', 'noise', 'neural', 'bit', 'implementation', 'source',
'design', 'gain', 'processor', 'synapse', 'device', 'array']
```

We can use perplexity and coherence scores as measures to evaluate the topic model. Typically, lower the perplexity, the better the model. Similarly, the lower the UMass score and the higher the **Cv** score in coherence, the better the model.

```
cv_coherence_model_lda = gensim.models.CoherenceModel(model=lda_model,
                                                        corpus=bow_corpus,
                                                   texts=norm_corpus_bigrams,
                                                       dictionary=dictionary,
                                                        coherence='c_v')
avg_coherence_cv = cv_coherence_model_lda.get_coherence()

umass_coherence_model_lda = gensim.models.CoherenceModel(model=lda_model,
                                                           corpus=bow_corpus,
                                                 texts=norm_corpus_bigrams,
                                                       dictionary=dictionary,
                                                          coherence='u_mass')
avg_coherence_umass = umass_coherence_model_lda.get_coherence()

perplexity = lda_model.log_perplexity(bow_corpus)

print('Avg. Coherence Score (Cv):', avg_coherence_cv)
print('Avg. Coherence Score (UMass):', avg_coherence_umass)
print('Model Perplexity:', perplexity)
```

```
Avg. Coherence Score (Cv): 0.47028476052247825
Avg. Coherence Score (UMass): -1.0433305600965896
Model Perplexity: -7.792233498252204
```

Not bad, but we have nothing to compare this against. Let's try to build another LDA topic model based on a separate package called MALLET, which has Gensim wrappers to make it easy to use from Python!

LDA Models with MALLET

The MALLET framework is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine learning applications to text. MALLET stands for *MAchine Learning for LanguagE Toolkit*. It was developed by Andrew McCallum along with several people at the University of Massachusetts Amherst. The MALLET topic modeling toolkit contains efficient, sampling-based implementations of Latent Dirichlet Allocation, Pachinko Allocation, and Hierarchical LDA. To use MALLET's capabilities, we need to download the framework.

```
!wget http://mallet.cs.umass.edu/dist/mallet-2.0.8.zip
```

```
--2018-11-08 20:06:13--  http://mallet.cs.umass.edu/dist/mallet-2.0.8.zip
Resolving mallet.cs.umass.edu (mallet.cs.umass.edu)... 128.119.246.70
Connecting to mallet.cs.umass.edu (mallet.cs.umass.
edu)|128.119.246.70|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16184794 (15M) [application/zip]
Saving to: 'mallet-2.0.8.zip'

mallet-2.0.8.zip    100%[===================>]  15.43M  1.35MB/s    in 12s

2018-11-08 20:06:25 (1.28 MB/s) - 'mallet-2.0.8.zip' saved
[16184794/16184794]
```

Windows users can download the package directly from the browser using the same URL mentioned in the preceding code. Once it's downloaded, we need to extract the contents from the archive.

```
!unzip -q mallet-2.0.8.zip
```

We are now ready to build our LDA model using MALLET. If you have multiple CPUs, Gensim can also use them for parallel processing and faster training.

```
MALLET_PATH = 'mallet-2.0.8/bin/mallet'
lda_mallet = gensim.models.wrappers.LdaMallet(mallet_path=MALLET_PATH,
                                              corpus=bow_corpus,
                                              num_topics=TOTAL_TOPICS,
                                              id2word=dictionary,
                                              iterations=500, workers=16)
```

We can now look at the generated topics by leveraging the following code snippet.

```
topics = [[(term, round(wt, 3))
               for term, wt in lda_mallet.show_topic(n, topn=20)]
                   for n in range(0, TOTAL_TOPICS)]

for idx, topic in enumerate(topics):
    print('Topic #'+str(idx+1)+':')
    print([term for term, wt in topic])
    print()
```

```
Topic #1:
['neuron', 'cell', 'response', 'stimulus', 'activity', 'pattern', 'signal',
'spike', 'effect', 'synaptic', 'frequency', 'neural', 'unit', 'connection',
'layer', 'cortical', 'firing', 'et_al', 'brain', 'temporal']

Topic #2:
['prediction', 'control', 'trajectory', 'target', 'task', 'expert',
'training', 'nonlinear', 'dynamic', 'linear', 'local', 'change',
'adaptive', 'mapping', 'hand', 'movement', 'controller', 'position',
'motor', 'architecture']

...,
...,

Topic #9:
['training', 'unit', 'pattern', 'hidden_unit', 'layer', 'net',
'classifier', 'class', 'training_set', 'classification', 'trained', 'test',
'task', 'back_propagation', 'hidden_layer', 'table', 'generalization',
'feature', 'size', 'architecture']

Topic #10:
['word', 'recognition', 'speech', 'sequence', 'feature', 'context',
'training', 'character', 'hmm', 'module', 'signal', 'letter', 'frame',
'trained', 'experiment', 'classification', 'architecture', 'speaker',
'window', 'class']
```

We can also evaluate our model using the perplexity and coherence metrics, as we did before.

```
cv_coherence_model_lda_mallet = gensim.models.CoherenceModel
                                              (model=lda_mallet,
                                              corpus=bow_corpus,
                                              texts=norm_corpus_bigrams,
                                              dictionary=dictionary,
                                              coherence='c_v')
avg_coherence_cv = cv_coherence_model_lda_mallet.get_coherence()
```

```
umass_coherence_model_lda_mallet = gensim.models.CoherenceModel
                                              (model=lda_mallet,
                                           corpus=bow_corpus,
                                           texts=norm_corpus_bigrams,
                                           dictionary=dictionary,
                                           coherence='u_mass')
avg_coherence_umass = umass_coherence_model_lda_mallet.get_coherence()
```

```
# from STDOUT: <500> LL/token: -8.53533
perplexity = -8.53533
print('Avg. Coherence Score (Cv):', avg_coherence_cv)
print('Avg. Coherence Score (UMass):', avg_coherence_umass)
print('Model Perplexity:', perplexity)
```

```
Avg. Coherence Score (Cv): 0.5008326905758488
Avg. Coherence Score (UMass): -1.0635635291342118
Model Perplexity: -8.53533
```

You can clearly see that the model from MALLET is much better based on these metrics as compared to the default LDA model from Gensim. Can we find the optimal number of topics that maximizes the coherence? This is a tough problem, but we can try doing it iteratively.

LDA Tuning: Finding the Optimal Number of Topics

Finding the optimal number of topics in a topic model is tough, given that it is like a model hyperparameter that you always have to set before training the model. We can use an iterative approach and build several models with differing numbers of topics and select the one that has the highest coherence score. To implement this method, we build the following function.

```
from tqdm import tqdm
```

```
def topic_model_coherence_generator(corpus, texts, dictionary,
                          start_topic_count=2, end_topic_count=10, step=1,
                                   cpus=1):
```

```
    models = []
    coherence_scores = []
    for topic_nums in tqdm(range(start_topic_count, end_topic_count+1, step)):
        mallet_lda_model = gensim.models.wrappers.LdaMallet
                                                   (mallet_path=MALLET_PATH,
                                              corpus=corpus,
                                              num_topics=topic_nums,
                                              id2word=dictionary,
                                              iterations=500, workers=cpus)
        cv_coherence_model_mallet_lda = gensim.models.CoherenceModel
                                               (model=mallet_lda_model,
                                                    corpus=corpus,
                                                    texts=texts,
                                                    dictionary=dictionary,
                                                    coherence='c_v')
        coherence_score = cv_coherence_model_mallet_lda.get_coherence()
        coherence_scores.append(coherence_score)
        models.append(mallet_lda_model)

    return models, coherence_scores
```

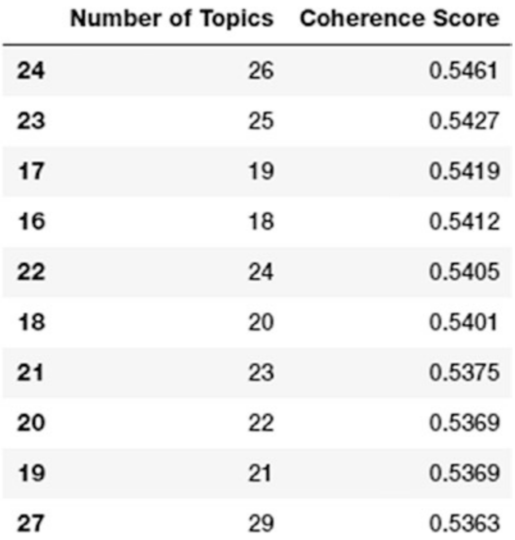
Let's put this function into action now and build several topic models, with the number of topics ranging from 2 to 30.

```
lda_models, coherence_scores = topic_model_coherence_generator(corpus=bow_corpus,
                                                texts=norm_corpus_bigrams,
                                                dictionary=dictionary,
                                                start_topic_count=2,
                                                end_topic_count=30, step=1,
                                                cpus=16)
```

```
100%|██████████████████████████████| 29/29 [37:48<00:00, 92.53s/it]
```

Note that this step might take some time to train, depending on your infrastructure since we will be training several topic models. One way to inspect the output is to sort the results by the coherence score and look at the number of topics. See Figure 6-6.

```
coherence_df = pd.DataFrame({'Number of Topics': range(2, 31, 1),
                             'Coherence Score': np.round(coherence_scores, 4)})
coherence_df.sort_values(by=['Coherence Score'], ascending=False).head(10)
```



	Number of Topics	Coherence Score
24	26	0.5461
23	25	0.5427
17	19	0.5419
16	18	0.5412
22	24	0.5405
18	20	0.5401
21	23	0.5375
20	22	0.5369
19	21	0.5369
27	29	0.5363

Figure 6-6. *Sorting the topic models based on the coherence score*

Let's plot a graph showing the number of topics per model and their corresponding coherence scores.

```
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
%matplotlib inline

x_ax = range(2, 31, 1)
y_ax = coherence_scores
plt.figure(figsize=(12, 6))
plt.plot(x_ax, y_ax, c='r')
plt.axhline(y=0.535, c='k', linestyle='--', linewidth=2)
plt.rcParams['figure.facecolor'] = 'white'
xl = plt.xlabel('Number of Topics')
yl = plt.ylabel('Coherence Score')
```

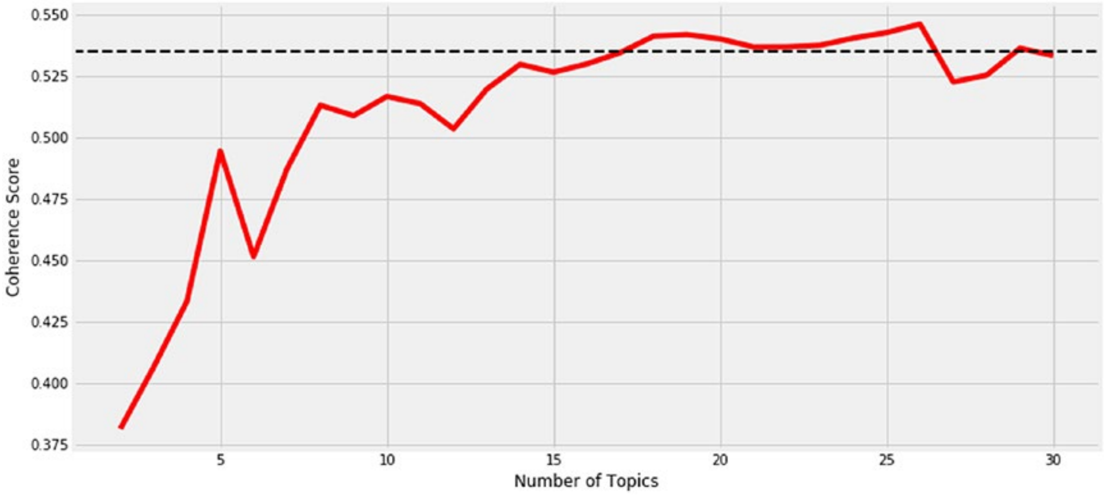


***Figure 6-7.** Topic model tuning the number of topics vs. coherence score*

From Figure 6-7, it looks like the score starts increasing rapidly when the number of topics is five and gradually starts plateauing at 19 or 20. We choose the optimal number of topics as 20, based on our intuition. We can retrieve the best model now:

```
best_model_idx = coherence_df[coherence_df['Number of Topics'] == 20].index[0]
best_lda_model = lda_models[best_model_idx]
best_lda_model.num_topics
```

```
20
```

Let's view all the 20 topics generated by our selected best model, similar to our previous models.

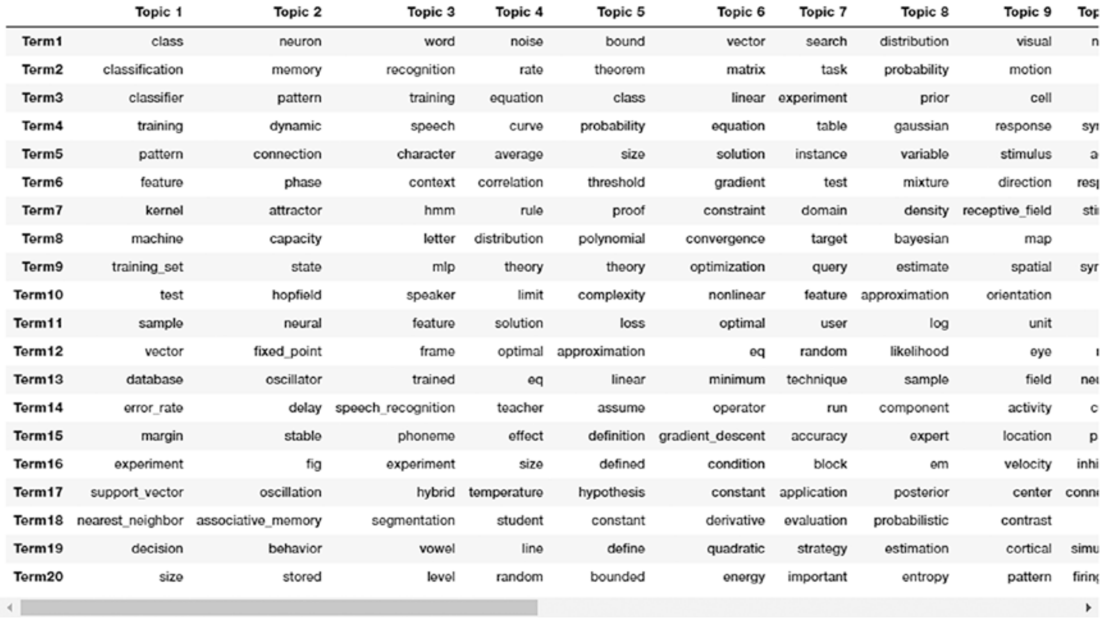
```
topics = [[(term, round(wt, 3))
               for term, wt in best_lda_model.show_topic(n, topn=20)]
                   for n in range(0, best_lda_model.num_topics)]

for idx, topic in enumerate(topics):
    print('Topic #'+str(idx+1)+':')
    print([term for term, wt in topic])
    print()
```

```
Topic #1:
['class', 'classification', 'classifier', 'training', 'pattern', 'feature',
'kernel', 'machine', 'training_set', 'test', 'sample', 'vector',
'database', 'error_rate', 'margin', 'experiment', 'support_vector',
'nearest_neighbor', 'decision', 'size']

...,

Topic #8:
['distribution', 'probability', 'prior', 'gaussian', 'variable', 'mixture',
'density', 'bayesian', 'estimate', 'approximation', 'log', 'likelihood',
'sample', 'component', 'expert', 'em', 'posterior', 'probabilistic',
'estimation', 'entropy']

Topic #9:
['visual', 'motion', 'cell', 'response', 'stimulus', 'direction', 'receptive_
field', 'map', 'spatial', 'orientation', 'unit', 'eye', 'field', 'activity',
'location', 'velocity', 'center', 'contrast', 'cortical', 'pattern']

...,

Topic #13:
['image', 'object', 'feature', 'pixel', 'face', 'view', 'recognition',
'representation', 'shape', 'scale', 'part', 'visual', 'region', 'position',
'scene', 'surface', 'vision', 'frame', 'texture', 'location']

Topic #14:
['control', 'action', 'state', 'policy', 'environment', 'controller',
'reinforcement_learning', 'task', 'optimal', 'robot', 'goal', 'step', 'reward',
'td', 'agent', 'adaptive', 'cost', 'reinforcement', 'trial', 'exploration']

...,

Topic #16:
['circuit', 'chip', 'current', 'analog', 'voltage', 'implementation',
'processor', 'bit', 'design', 'device', 'computation', 'parallel', 'digital',
'operation', 'array', 'neural', 'synapse', 'element', 'hardware', 'transistor']
```

```
Topic #17:
['rule', 'representation', 'module', 'structure', 'human', 'movement',
'motor', 'target', 'language', 'subject', 'connectionist', 'position', 'task',
'context', 'trajectory', 'hand', 'role', 'symbol', 'learned', 'theory']

Topic #18:
['vector', 'map', 'distance', 'cluster', 'local', 'dimension',
'clustering', 'mapping', 'dimensional', 'region', 'structure', 'center',
'rbf', 'pca', 'basis_function', 'linear', 'representation', 'global',
'principal_component', 'projection']

Topic #19:
['signal', 'filter', 'frequency', 'source', 'channel', 'noise', 'component',
'response', 'temporal', 'sound', 'auditory', 'detection', 'phase', 'ica',
'adaptation', 'amplitude', 'subject', 'eeg', 'change', 'correlation']

Topic #20:
['prediction', 'training', 'estimate', 'regression', 'test', 'noise',
'selection', 'variance', 'training_set', 'sample', 'ensemble',
'estimation', 'average', 'nonlinear', 'linear', 'estimator', 'cross_
validation', 'pruning', 'bias', 'risk']
```

A better way of visualizing the topics is to build a term-topic dataframe, as depicted in Figure 6-8.

```
topics_df = pd.DataFrame([[term for term, wt in topic]
                              for topic in topics],
                         columns = ['Term'+str(i) for i in range(1, 21)],
                         index=['Topic '+str(t) for t in range(1, best_lda_
                         model.num_topics+1)]).T
topics_df
```

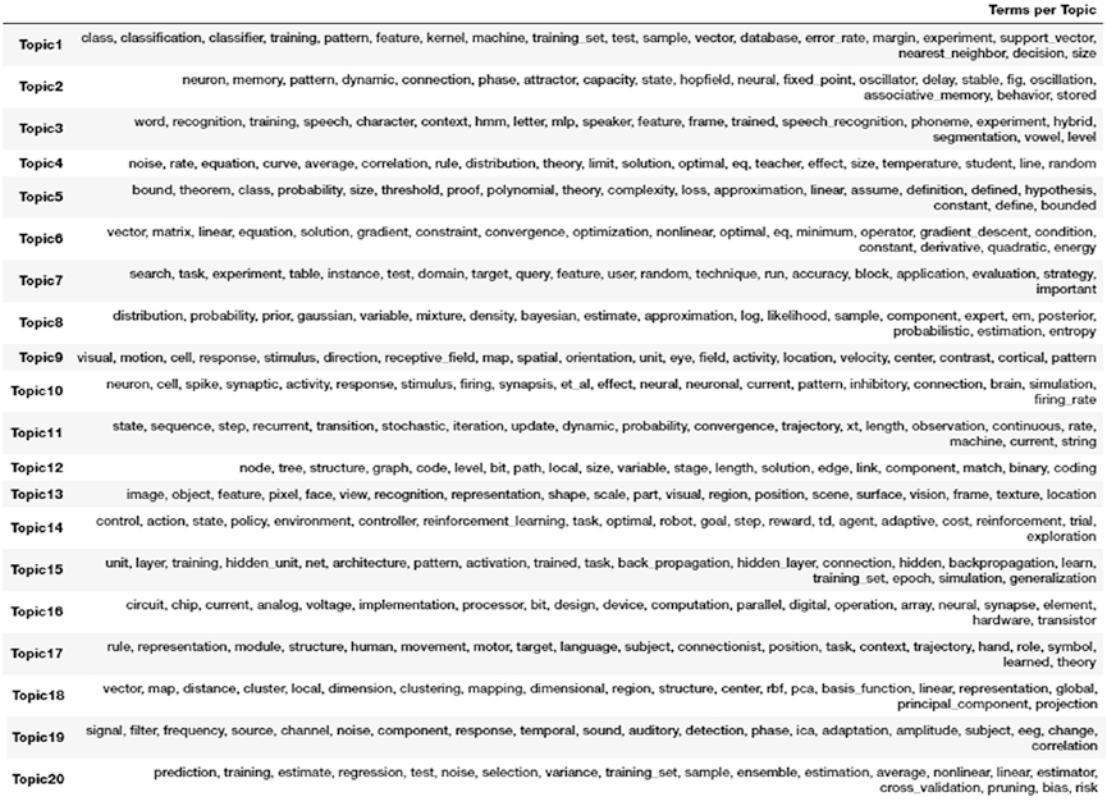


	Topic 1	Topic 2	Topic 3	Topic 4	Topic 5	Topic 6	Topic 7	Topic 8	Topic 9	Top
Term1	class	neuron	word	noise	bound	vector	search	distribution	visual	n
Term2	classification	memory	recognition	rate	theorem	matrix	task	probability	motion	
Term3	classifier	pattern	training	equation	class	linear	experiment	prior	cell	
Term4	training	dynamic	speech	curve	probability	equation	table	gaussian	response	syı
Term5	pattern	connection	character	average	size	solution	instance	variable	stimulus	a
Term6	feature	phase	context	correlation	threshold	gradient	test	mixture	direction	res|
Term7	kernel	attractor	hmm	rule	proof	constraint	domain	density	receptive_field	sti
Term8	machine	capacity	letter	distribution	polynomial	convergence	target	bayesian	map	
Term9	training_set	state	mlp	theory	theory	optimization	query	estimate	spatial	syr
Term10	test	hopfield	speaker	limit	complexity	nonlinear	feature	approximation	orientation	
Term11	sample	neural	feature	solution	loss	optimal	user	log	unit	
Term12	vector	fixed_point	frame	optimal	approximation	eq	random	likelihood	eye	ı
Term13	database	oscillator	trained	eq	linear	minimum	technique	sample	field	neı
Term14	error_rate	delay	speech_recognition	teacher	assume	operator	run	component	activity	c
Term15	margin	stable	phoneme	effect	definition	gradient_descent	accuracy	expert	location	p
Term16	experiment	fig	experiment	size	defined	condition	block	em	velocity	inhi
Term17	support_vector	oscillation	hybrid	temperature	hypothesis	constant	application	posterior	center	conn
Term18	nearest_neighbor	associative_memory	segmentation	student	constant	derivative	evaluation	probabilistic	contrast	
Term19	decision	behavior	vowel	line	define	quadratic	strategy	estimation	cortical	simu
Term20	size	stored	level	random	bounded	energy	important	entropy	pattern	firin

Figure 6-8. *Generated topics from our LDA topic model*

Another easy way to view the topics is to create a topic-term dataframe, whereby each topic is represented in a row with the terms of the topic being represented as a comma-separated string.

```
pd.set_option('display.max_colwidth', -1)
topics_df = pd.DataFrame([', '.join([term for term, wt in topic])
                              for topic in topics],
                         columns = ['Terms per Topic'],
                         index=['Topic'+str(t) for t in range(1, best_lda_
                         model.num_topics+1)]
                         )
topics_df
```



	Terms per Topic
Topic1	class, classification, classifier, training, pattern, feature, kernel, machine, training_set, test, sample, vector, database, error_rate, margin, experiment, support_vector, nearest_neighbor, decision, size
Topic2	neuron, memory, pattern, dynamic, connection, phase, attractor, capacity, state, hopfield, neural, fixed_point, oscillator, delay, stable, fig, oscillation, associative_memory, behavior, stored
Topic3	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level
Topic4	noise, rate, equation, curve, average, correlation, rule, distribution, theory, limit, solution, optimal, eq, teacher, effect, size, temperature, student, line, random
Topic5	bound, theorem, class, probability, size, threshold, proof, polynomial, theory, complexity, loss, approximation, linear, assume, definition, defined, hypothesis, constant, define, bounded
Topic6	vector, matrix, linear, equation, solution, gradient, constraint, convergence, optimization, nonlinear, optimal, eq, minimum, operator, gradient_descent, condition, constant, derivative, quadratic, energy
Topic7	search, task, experiment, table, instance, test, domain, target, query, feature, user, random, technique, run, accuracy, block, application, evaluation, strategy, important
Topic8	distribution, probability, prior, gaussian, variable, mixture, density, bayesian, estimate, approximation, log, likelihood, sample, component, expert, em, posterior, probabilistic, estimation, entropy
Topic9	visual, motion, cell, response, stimulus, direction, receptive_field, map, spatial, orientation, unit, eye, field, activity, location, velocity, center, contrast, cortical, pattern
Topic10	neuron, cell, spike, synaptic, activity, response, stimulus, firing, synapsis, et_al, effect, neural, neuronal, current, pattern, inhibitory, connection, brain, simulation, firing_rate
Topic11	state, sequence, step, recurrent, transition, stochastic, iteration, update, dynamic, probability, convergence, trajectory, xt, length, observation, continuous, rate, machine, current, string
Topic12	node, tree, structure, graph, code, level, bit, path, local, size, variable, stage, length, solution, edge, link, component, match, binary, coding
Topic13	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location
Topic14	control, action, state, policy, environment, controller, reinforcement_learning, task, optimal, robot, goal, step, reward, td, agent, adaptive, cost, reinforcement, trial, exploration
Topic15	unit, layer, training, hidden_unit, net, architecture, pattern, activation, trained, task, back_propagation, hidden_layer, connection, hidden, backpropagation, learn, training_set, epoch, simulation, generalization
Topic16	circuit, chip, current, analog, voltage, implementation, processor, bit, design, device, computation, parallel, digital, operation, array, neural, synapse, element, hardware, transistor
Topic17	rule, representation, module, structure, human, movement, motor, target, language, subject, connectionist, position, task, context, trajectory, hand, role, symbol, learned, theory
Topic18	vector, map, distance, cluster, local, dimension, clustering, mapping, dimensional, region, structure, center, rbf, pca, basis_function, linear, representation, global, principal_component, projection
Topic19	signal, filter, frequency, source, channel, noise, component, response, temporal, sound, auditory, detection, phase, ica, adaptation, amplitude, subject, eeg, change, correlation
Topic20	prediction, training, estimate, regression, test, noise, selection, variance, training_set, sample, ensemble, estimation, average, nonlinear, linear, estimator, cross_validation, pruning, bias, risk

Figure 6-9. *Viewing all the topics of our LDA topic model*

The dataframe depicted in Figure 6-9 gives us an easy way to visualize and understand the major themes in our corpus of research papers. Do you notice any interesting patterns? I observed some very interesting themes around neural networks, signal processing, dimension reduction, reinforcement learning, neural models in chips, and image and visual recognition!

Interpreting Topic Model Results

Let's look at some interesting ways of diving deeper and interpreting results from our topic model. An interesting point to remember is, given a corpus of documents (in the form of features, e.g., Bag of Words) and a trained topic model, you can predict the distribution of topics in each document (research paper in this case) with the following code.

```
tm_results = best_lda_model[bow_corpus]
```

We can now get the most dominant topic per research paper with some intelligent sorting and indexing using the following code.

```
corpus_topics = [sorted(topics, key=lambda record: -record[1])[0]
                     for topics in tm_results]
corpus_topics[:5]
```

```
[(16, 0.2115988756613756),
 (5, 0.29989652050187554),
 (9, 0.3307915758896151),
 (8, 0.5447463768115942),
 (9, 0.18093823158652983)]
```

This provides a plethora of options that can be leveraged to extract useful insights from our corpus of research papers. To enable this, we construct a master dataframe that will hold the base statistics, which we use soon to depict different useful insights.

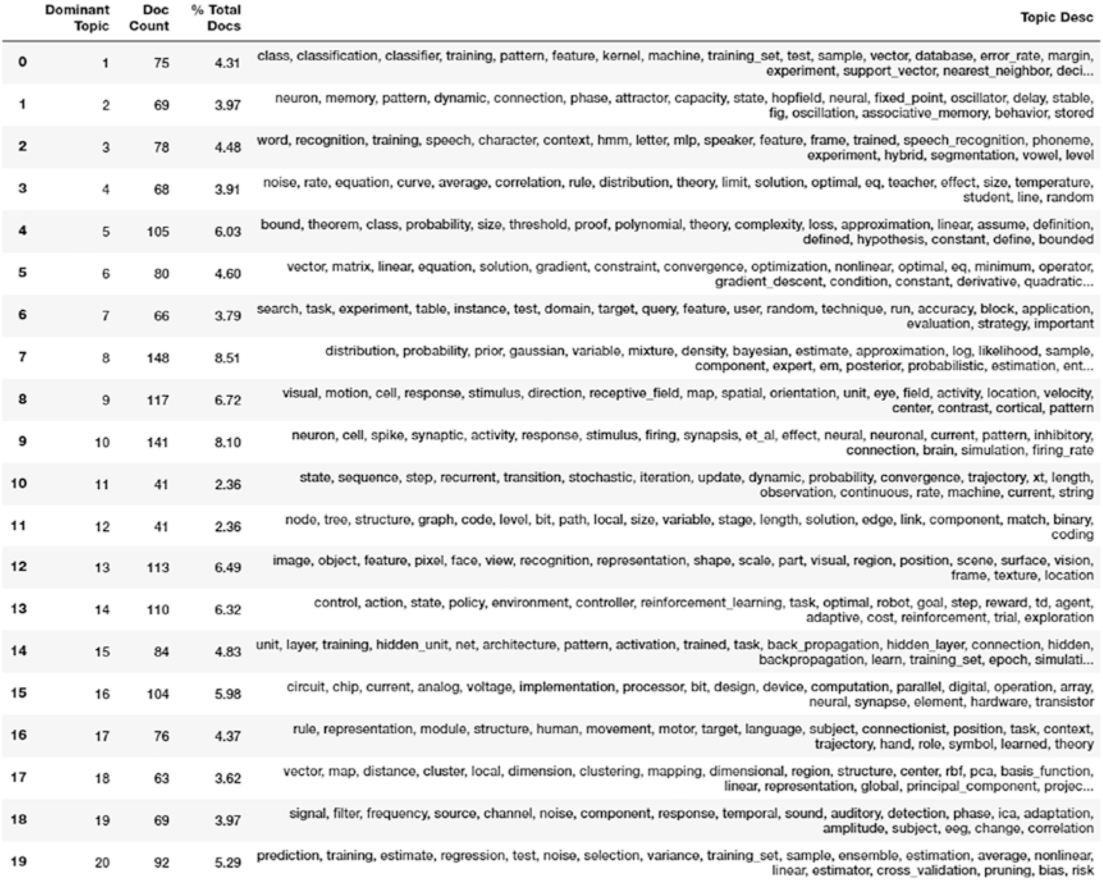
```
corpus_topic_df = pd.DataFrame()
corpus_topic_df['Document'] = range(0, len(papers))
corpus_topic_df['Dominant Topic'] = [item[0]+1 for item in corpus_topics]
corpus_topic_df['Contribution %'] = [round(item[1]*100, 2) for item in
corpus_topics]
corpus_topic_df['Topic Desc'] = [topics_df.iloc[t[0]]['Terms per Topic']
for t in corpus_topics]
corpus_topic_df['Paper'] = papers
```

Let's now take a look at various ways we can transform these results and extract meaningful insights from our research papers and their topics.

Dominant Topics Distribution Across Corpus

The first thing we can do is look at the overall distribution of each topic across the corpus of research papers. Mainly we want to determine the total number of papers and the total percentage of papers where each of the 20 topics was the most dominant.

```
pd.set_option('display.max_colwidth', 200)
topic_stats_df = corpus_topic_df.groupby('Dominant Topic').agg({
                                              'Dominant Topic': {
                                                  'Doc Count': np.size,
                                          '% Total Docs': np.size }
                                      })
topic_stats_df = topic_stats_df['Dominant Topic'].reset_index()
topic_stats_df['% Total Docs'] = topic_stats_df['% Total Docs'].
apply(lambda row: round((row*100) / len(papers), 2))
topic_stats_df['Topic Desc'] = [topics_df.iloc[t]['Terms per Topic'] for t in
range(len(topic_stats_df))]
topic_stats_df
```



	Dominant Topic	Doc Count	% Total Docs	Topic Desc
0	1	75	4.31	class, classification, classifier, training, pattern, feature, kernel, machine, training_set, test, sample, vector, database, error_rate, margin, experiment, support_vector, nearest_neighbor, deci...
1	2	69	3.97	neuron, memory, pattern, dynamic, connection, phase, attractor, capacity, state, hopfield, neural, fixed_point, oscillator, delay, stable, fig, oscillation, associative_memory, behavior, stored
2	3	78	4.48	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level
3	4	68	3.91	noise, rate, equation, curve, average, correlation, rule, distribution, theory, limit, solution, optimal, eq, teacher, effect, size, temperature, student, line, random
4	5	105	6.03	bound, theorem, class, probability, size, threshold, proof, polynomial, theory, complexity, loss, approximation, linear, assume, definition, defined, hypothesis, constant, define, bounded
5	6	80	4.60	vector, matrix, linear, equation, solution, gradient, constraint, convergence, optimization, nonlinear, optimal, eq, minimum, operator, gradient_descent, condition, constant, derivative, quadratic...
6	7	66	3.79	search, task, experiment, table, instance, test, domain, target, query, feature, user, random, technique, run, accuracy, block, application, evaluation, strategy, important
7	8	148	8.51	distribution, probability, prior, gaussian, variable, mixture, density, bayesian, estimate, approximation, log, likelihood, sample, component, expert, em, posterior, probabilistic, estimation, ent...
8	9	117	6.72	visual, motion, cell, response, stimulus, direction, receptive_field, map, spatial, orientation, unit, eye, field, activity, location, velocity, center, contrast, cortical, pattern
9	10	141	8.10	neuron, cell, spike, synaptic, activity, response, stimulus, firing, synapsis, et_al, effect, neural, neuronal, current, pattern, inhibitory, connection, brain, simulation, firing_rate
10	11	41	2.36	state, sequence, step, recurrent, transition, stochastic, iteration, update, dynamic, probability, convergence, trajectory, xt, length, observation, continuous, rate, machine, current, string
11	12	41	2.36	node, tree, structure, graph, code, level, bit, path, local, size, variable, stage, length, solution, edge, link, component, match, binary, coding
12	13	113	6.49	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location
13	14	110	6.32	control, action, state, policy, environment, controller, reinforcement_learning, task, optimal, robot, goal, step, reward, td, agent, adaptive, cost, reinforcement, trial, exploration
14	15	84	4.83	unit, layer, training, hidden_unit, net, architecture, pattern, activation, trained, task, back_propagation, hidden_layer, connection, hidden, backpropagation, learn, training_set, epoch, simulati...
15	16	104	5.98	circuit, chip, current, analog, voltage, implementation, processor, bit, design, device, computation, parallel, digital, operation, array, neural, synapse, element, hardware, transistor
16	17	76	4.37	rule, representation, module, structure, human, movement, motor, target, language, subject, connectionist, position, task, context, trajectory, hand, role, symbol, learned, theory
17	18	63	3.62	vector, map, distance, cluster, local, dimension, clustering, mapping, dimensional, region, structure, center, rbf, pca, basis_function, linear, representation, global, principal_component, projec...
18	19	69	3.97	signal, filter, frequency, source, channel, noise, component, response, temporal, sound, auditory, detection, phase, ica, adaptation, amplitude, subject, eeg, change, correlation
19	20	92	5.29	prediction, training, estimate, regression, test, noise, selection, variance, training_set, sample, ensemble, estimation, average, nonlinear, linear, estimator, cross_validation, pruning, bias, risk

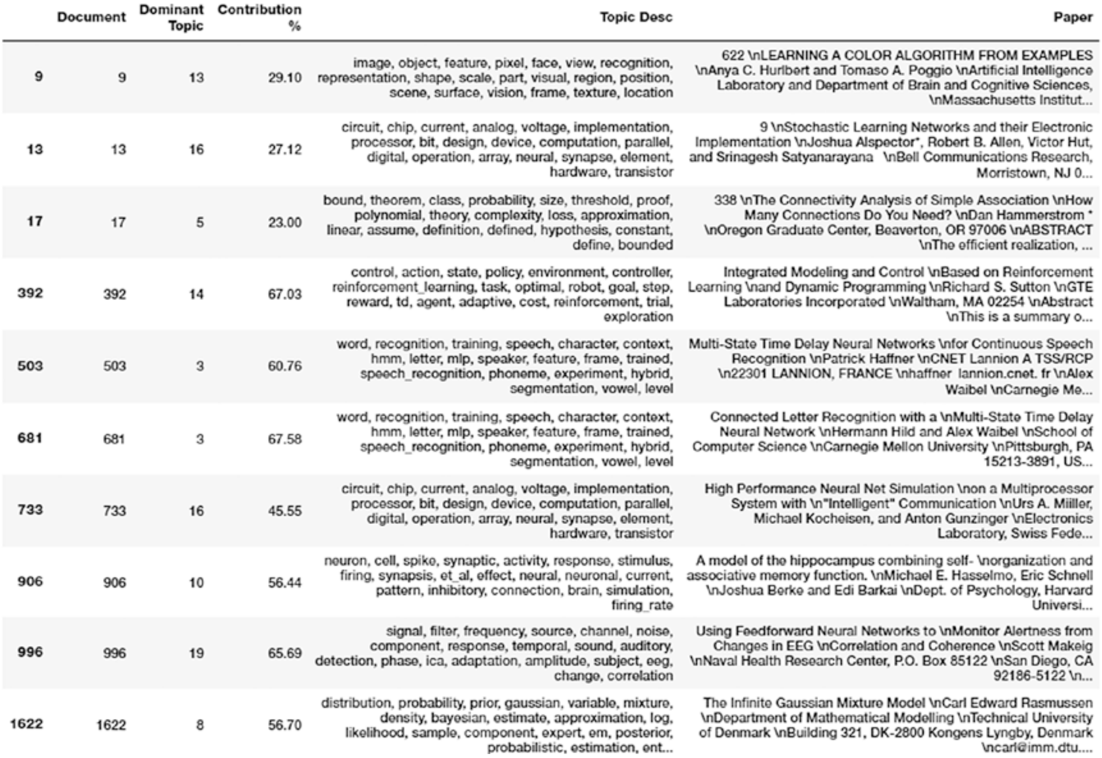
Figure 6-10. *Viewing the distribution of dominant topics*

The results in Figure 6-10 show us that most of the papers cover topics of probabilistic models and Bayesian modeling (Topic #8), followed by papers covering modeling and simulating how the brain works with neurons, cells, stimulus, and connections (Topic #10). Even Topic #14, covering reinforcement learning and robotics, has almost 6.32% representation of the total number of papers. This tells us it's not a new thing and people have been researching it for decades!

Dominant Topics in Specific Research Papers

Another interesting perspective is to select specific papers, view the most dominant topic in each of those papers, and see if that makes sense.

```
pd.set_option('display.max_colwidth', 200)
(corpus_topic_df[corpus_topic_df['Document']
                    .isin([681, 9, 392, 1622, 17,
                            906, 996, 503, 13, 733])])
```



	Document	Dominant Topic	Contribution %	Topic Desc	Paper
9	9	13	29.10	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location	622 \nLEARNING A COLOR ALGORITHM FROM EXAMPLES \nAnya C. Hurlbert and Tomaso A. Poggio \nArtificial Intelligence Laboratory and Department of Brain and Cognitive Sciences, \nMassachusetts Institut...
13	13	16	27.12	circuit, chip, current, analog, voltage, implementation, processor, bit, design, device, computation, parallel, digital, operation, array, neural, synapse, element, hardware, transistor	9 \nStochastic Learning Networks and their Electronic Implementation \nJoshua Alspector*, Robert B. Allen, Victor Hut, and Srinagesh Satyanarayana \nBell Communications Research, Morristown, NJ 0...
17	17	5	23.00	bound, theorem, class, probability, size, threshold, proof, polynomial, theory, complexity, loss, approximation, linear, assume, definition, defined, hypothesis, constant, define, bounded	338 \nThe Connectivity Analysis of Simple Association \nHow Many Connections Do You Need? \nDan Hammerstrom * \nOregon Graduate Center, Beaverton, OR 97006 \nABSTRACT \nThe efficient realization, ...
392	392	14	67.03	control, action, state, policy, environment, controller, reinforcement_learning, task, optimal, robot, goal, step, reward, td, agent, adaptive, cost, reinforcement, trial, exploration	Integrated Modeling and Control \nBased on Reinforcement Learning \nand Dynamic Programming \nRichard S. Sutton \nGTE Laboratories Incorporated \nWaltham, MA 02254 \nAbstract \nThis is a summary o...
503	503	3	60.76	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level	Multi-State Time Delay Neural Networks \nfor Continuous Speech Recognition \nPatrick Haffner \nCNET Lannion A TSS/RCP \n22301 LANNION, FRANCE \nhaffner lannion.cnet. fr \nAlex Waibel \nCarnegie Me...
681	681	3	67.58	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level	Connected Letter Recognition with a \nMulti-State Time Delay Neural Network \nHermann Hild and Alex Waibel \nSchool of Computer Science \nCarnegie Mellon University \nPittsburgh, PA 15213-3891, US...
733	733	16	45.55	circuit, chip, current, analog, voltage, implementation, processor, bit, design, device, computation, parallel, digital, operation, array, neural, synapse, element, hardware, transistor	High Performance Neural Net Simulation \non a Multiprocessor System with \n"Intelligent" Communication \nUrs A. Miiller, Michael Kocheisen, and Anton Gunzinger \nElectronics Laboratory, Swiss Fede...
906	906	10	56.44	neuron, cell, spike, synaptic, activity, response, stimulus, firing, synapsis, et_al, effect, neural, neuronal, current, pattern, inhibitory, connection, brain, simulation, firing_rate	A model of the hippocampus combining self- \norganization and associative memory function. \nMichael E. Hasselmo, Eric Schnell \nJoshua Berke and Edi Barkai \nDept. of Psychology, Harvard Universi...
996	996	19	65.69	signal, filter, frequency, source, channel, noise, component, response, temporal, sound, auditory, detection, phase, ica, adaptation, amplitude, subject, eeg, change, correlation	Using Feedforward Neural Networks to \nMonitor Alertness from Changes in EEG \nCorrelation and Coherence \nScott Makeig \nNaval Health Research Center, P.O. Box 85122 \nSan Diego, CA 92186-5122 \n...
1622	1622	8	56.70	distribution, probability, prior, gaussian, variable, mixture, density, bayesian, estimate, approximation, log, likelihood, sample, component, expert, em, posterior, probabilistic, estimation, ent...	The Infinite Gaussian Mixture Model \nCarl Edward Rasmussen \nDepartment of Mathematical Modelling \nTechnical University of Denmark \nBuilding 321, DK-2800 Kongens Lyngby, Denmark \ncarl@imm.dtu....

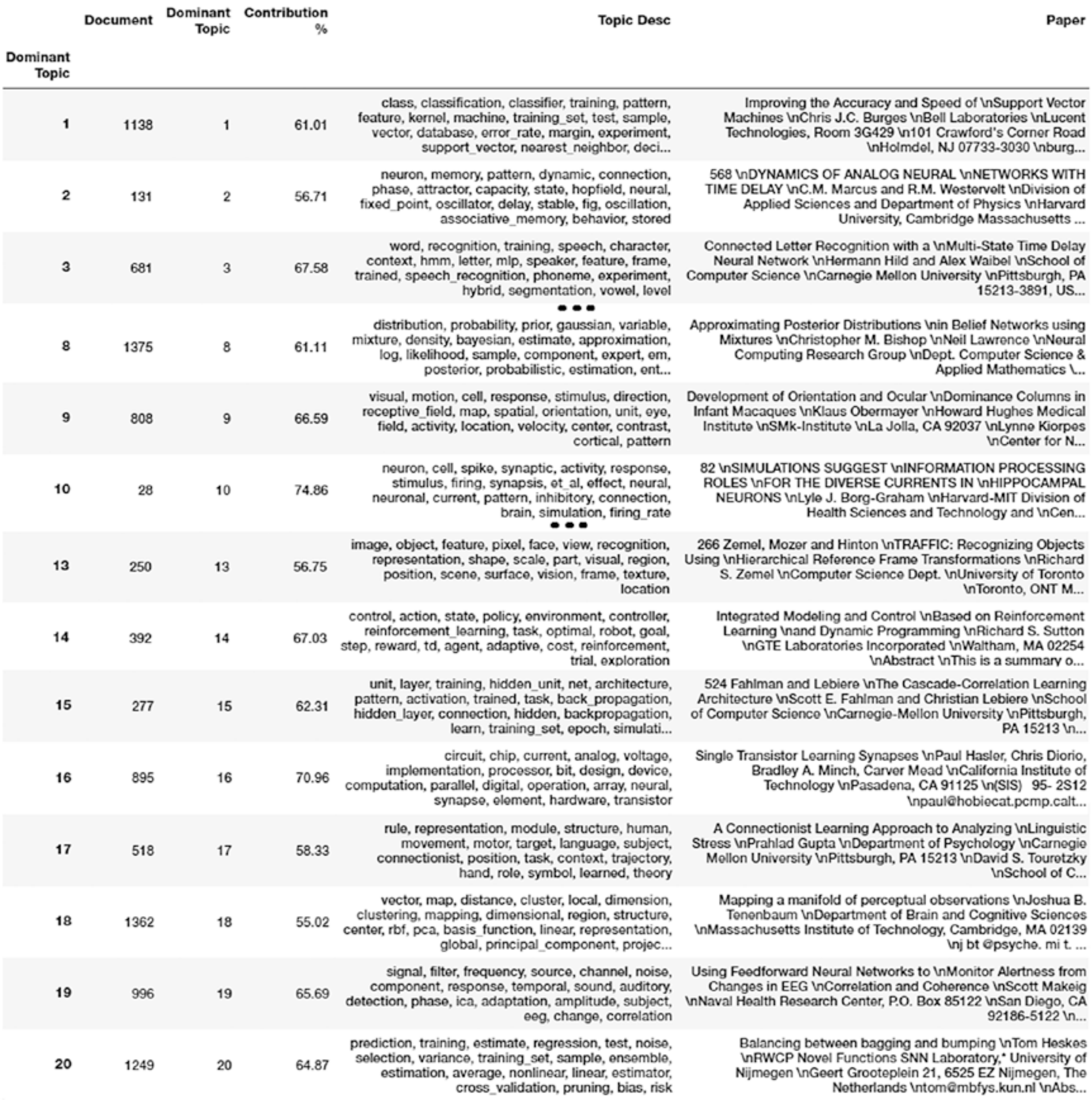
Figure 6-11. *Viewing the dominance of topics in research papers*

Based on the results in Figure 6-11, we can see that they make perfect sense! Papers on reinforcement learning, signal processing, gaussian mixture models, processor simulations, word recognitions, and many more have corresponding relevant topics as the most dominant topics. This tells us that our topic model is working well.

Relevant Research Papers per Topic Based on Dominance

A better way of representation is to try to retrieve the corresponding research paper that has the highest representation for each of the 20 topics.

```
corpus_topic_df.groupby('Dominant Topic').apply(lambda topic_set:
                                                   (topic_set.sort_
                                                    values(by=['Contribution %'],
                                                    ascending=False).iloc[0]))
```



	Document	Dominant Topic	Contribution %	Topic Desc	Paper
Dominant Topic					
1	1138	1	61.01	class, classification, classifier, training, pattern, feature, kernel, machine, training_set, test, sample, vector, database, error_rate, margin, experiment, support_vector, nearest_neighbor, deci...	Improving the Accuracy and Speed of \nSupport Vector Machines \nChris J.C. Burges \nBell Laboratories \nLucent Technologies, Room 3G429 \n101 Crawford's Corner Road \nHolmdel, NJ 07733-3030 \nburg...
2	131	2	56.71	neuron, memory, pattern, dynamic, connection, phase, attractor, capacity, state, hopfield, neural, fixed_point, oscillator, delay, stable, fig, oscillation, associative_memory, behavior, stored	568 \nDYNAMICS OF ANALOG NEURAL \nNETWORKS WITH TIME DELAY \nC.M. Marcus and R.M. Westervelt \nDivision of Applied Sciences and Department of Physics \nHarvard University, Cambridge Massachusetts ...
3	681	3	67.58	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level	Connected Letter Recognition with a \nMulti-State Time Delay Neural Network \nHermann Hild and Alex Waibel \nSchool of Computer Science \nCarnegie Mellon University \nPittsburgh, PA 15213-3891, US...
				•••	
8	1375	8	61.11	distribution, probability, prior, gaussian, variable, mixture, density, bayesian, estimate, approximation, log, likelihood, sample, component, expert, em, posterior, probabilistic, estimation, ent...	Approximating Posterior Distributions \nin Belief Networks using Mixtures \nChristopher M. Bishop \nNeil Lawrence \nNeural Computing Research Group \nDept. Computer Science & Applied Mathematics \...
9	808	9	66.59	visual, motion, cell, response, stimulus, direction, receptive_field, map, spatial, orientation, unit, eye, field, activity, location, velocity, center, contrast, cortical, pattern	Development of Orientation and Ocular \nDominance Columns in Infant Macaques \nKlaus Obermayer \nHoward Hughes Medical Institute \nSMk-Institute \nLa Jolla, CA 92037 \nLynne Kiorpes \nCenter for N...
10	28	10	74.86	neuron, cell, spike, synaptic, activity, response, stimulus, firing, synapsis, et_al, effect, neural, neuronal, current, pattern, inhibitory, connection, brain, simulation, firing_rate	82 \nSIMULATIONS SUGGEST \nINFORMATION PROCESSING ROLES \nFOR THE DIVERSE CURRENTS IN \nHIPPOCAMPAL NEURONS \nLyle J. Borg-Graham \nHarvard-MIT Division of Health Sciences and Technology and \nCen...
				•••	
13	250	13	56.75	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location	266 Zemel, Mozer and Hinton \nTRAFFIC: Recognizing Objects Using \nHierarchical Reference Frame Transformations \nRichard S. Zemel \nComputer Science Dept. \nUniversity of Toronto \nToronto, ONT M...
14	392	14	67.03	control, action, state, policy, environment, controller, reinforcement_learning, task, optimal, robot, goal, step, reward, td, agent, adaptive, cost, reinforcement, trial, exploration	Integrated Modeling and Control \nBased on Reinforcement Learning \nand Dynamic Programming \nRichard S. Sutton \nGTE Laboratories Incorporated \nWaltham, MA 02254 \nAbstract \nThis is a summary o...
15	277	15	62.31	unit, layer, training, hidden_unit, net, architecture, pattern, activation, trained, task, back_propagation, hidden_layer, connection, hidden, backpropagation, learn, training_set, epoch, simulati...	524 Fahlman and Lebiere \nThe Cascade-Correlation Learning Architecture \nScott E. Fahlman and Christian Lebiere \nSchool of Computer Science \nCarnegie-Mellon University \nPittsburgh, PA 15213 \n...
16	895	16	70.96	circuit, chip, current, analog, voltage, implementation, processor, bit, design, device, computation, parallel, digital, operation, array, neural, synapse, element, hardware, transistor	Single Transistor Learning Synapses \nPaul Hasler, Chris Diorio, Bradley A. Minch, Carver Mead \nCalifornia Institute of Technology \nPasadena, CA 91125 \n(SIS) 95- 2S12 \npaul@hobiecat.pcmp.calt...
17	518	17	58.33	rule, representation, module, structure, human, movement, motor, target, language, subject, connectionist, position, task, context, trajectory, hand, role, symbol, learned, theory	A Connectionist Learning Approach to Analyzing \nLinguistic Stress \nPrahlad Gupta \nDepartment of Psychology \nCarnegie Mellon University \nPittsburgh, PA 15213 \nDavid S. Touretzky \nSchool of C...
18	1362	18	55.02	vector, map, distance, cluster, local, dimension, clustering, mapping, dimensional, region, structure, center, rbf, pca, basis_function, linear, representation, global, principal_component, projec...	Mapping a manifold of perceptual observations \nJoshua B. Tenenbaum \nDepartment of Brain and Cognitive Sciences \nMassachusetts Institute of Technology, Cambridge, MA 02139 \nj bt @psyche. mi t. ...
19	996	19	65.69	signal, filter, frequency, source, channel, noise, component, response, temporal, sound, auditory, detection, phase, ica, adaptation, amplitude, subject, eeg, change, correlation	Using Feedforward Neural Networks to \nMonitor Alertness from Changes in EEG \nCorrelation and Coherence \nScott Makeig \nNaval Health Research Center, P.O. Box 85122 \nSan Diego, CA 92186-5122 \n...
20	1249	20	64.87	prediction, training, estimate, regression, test, noise, selection, variance, training_set, sample, ensemble, estimation, average, nonlinear, linear, estimator, cross_validation, pruning, bias, risk	Balancing between bagging and bumping \nTom Heskes \nRWCP Novel Functions SNN Laboratory,* University of Nijmegen \nGeert Grooteplein 21, 6525 EZ Nijmegen, The Netherlands \ntom@mbfys.kun.nl \nAbs...

Figure 6-12. *Viewing each topic and corresponding paper with its maximum contribution*

We do not show all the topics in Figure 6-12 due to space constraints, but you get the idea and you can view the entire output in the Jupyter notebook. You can even open each of the research papers based on the index and read the full contents to see if it makes sense! Based on the paper titles and the corresponding topics depicted in Figure 6-12, they do make sense. It looks like our model has captured the relevant latent patterns and themes in our corpus.

Predicting Topics for New Research Papers

Even though topic models are unsupervised models, we can estimate or predict potential topics for new documents based on what it has learned previously on the so-called "training" corpus. For testing our model, I have manually downloaded some recent papers from the NIPS 16 conference proceedings. Testing our model on these papers is going to be an interesting exercise.

```
import glob
# papers manually downloaded from NIPS 16
# https://papers.nips.cc/book/advances-in-neural-information-processing-
systems-29-2016

new_paper_files = glob.glob('nips16*.txt')
new_papers = []
for fn in new_paper_files:
    with open(fn, encoding='utf-8', errors='ignore', mode='r+') as f:
        data = f.read()
        new_papers.append(data)

print('Total New Papers:', len(new_papers))

Total New Papers: 4
```

You will find the papers downloaded in the corresponding folder for this chapter in the GitHub repository for this book at https://github.com/dipanjanS/text-analytics-with-python.

We need to build a text wrangling and feature engineering pipeline, which should match the same steps we followed when training our topic model.

```
def text_preprocessing_pipeline(documents, normalizer_fn, bigram_model):
    norm_docs = normalizer_fn(documents)
    norm_docs_bigrams = bigram_model[norm_docs]
    return norm_docs_bigrams

def bow_features_pipeline(tokenized_docs, dictionary):
    paper_bow_features = [dictionary.doc2bow(text)
                              for text in tokenized_docs]
    return paper_bow_features
```

```
norm_new_papers = text_preprocessing_pipeline(documents=new_papers,
                                              normalizer_fn=normalize_corpus,
                                              bigram_model=bigram_model)
norm_bow_features = bow_features_pipeline(tokenized_docs=norm_new_papers,
                                          dictionary=dictionary)
```

We can now validate if the transformations worked with the following code.

```
print(norm_new_papers[0][:30])
```

```
['cooperative', 'graphical_model', 'josip', 'djolonga', 'dept_
computer', 'science', 'eth', 'zurich', 'josipd', 'inf', 'ethz', 'ch',
'stefanie', 'jegelka', 'csail', 'mit', 'stefje', 'mit_edu', 'sebastian',
'tschiatschek', 'dept_computer', 'science', 'eth', 'zurich', 'stschia',
'inf', 'ethz', 'ch', 'andreas', 'krause']
```

```
print(norm_bow_features[0][:30])
```

```
[(0, 1), (1, 1), (6, 1), (17, 1), (18, 1), (19, 1), (25, 1), (31, 2), (36,
2), (38, 1), (39, 17), (41, 3), (43, 1), (45, 1), (49, 2), (50, 4), (51,
1), (52, 2), (54, 1), (60, 1), (65, 1), (66, 3), (68, 7), (71, 8), (76, 4),
(77, 2), (87, 1), (88, 3), (105, 1), (106, 1)]
```

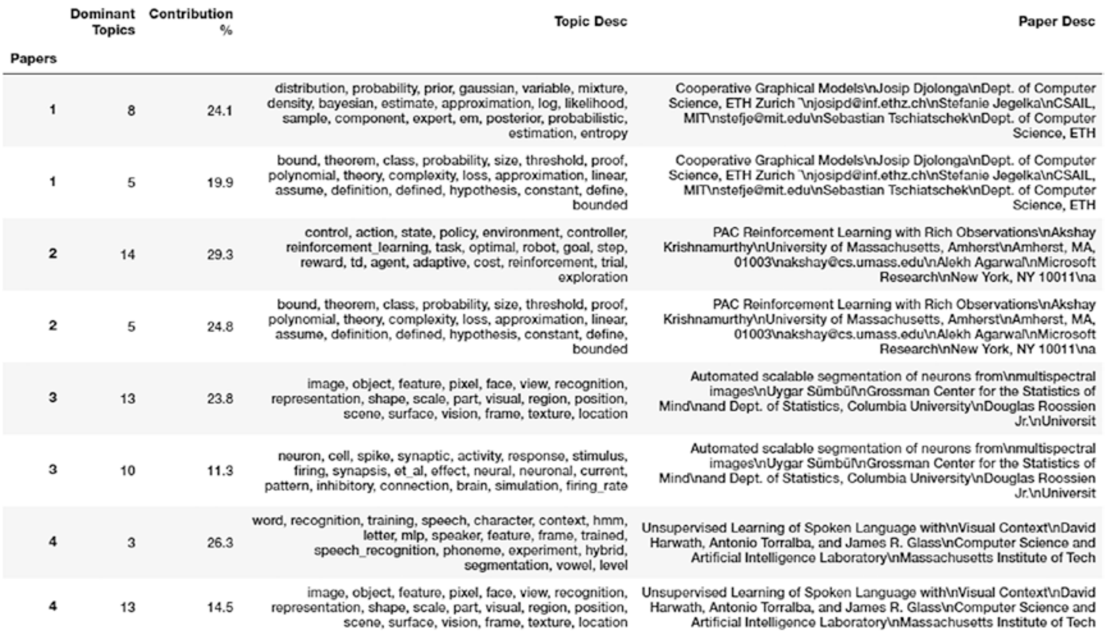
Let's now build a generic function that can extract the top **N** topics from any research paper using our trained model.

```
def get_topic_predictions(topic_model, corpus, topn=3):
    topic_predictions = topic_model[corpus]
    best_topics = [[(topic, round(wt, 3))
                        for topic, wt in sorted(topic_predictions[i],
                                                key=lambda row: -row[1])
                                                [:topn]]
                            for i in range(len(topic_predictions))]
    return best_topics

# putting the function in action
topic_preds = get_topic_predictions(topic_model=best_lda_model,
                                    corpus=norm_bow_features, topn=2)
topic_preds
```

```
[[(7, 0.241), (4, 0.199)],
 [(13, 0.293), (4, 0.248)],
 [(12, 0.238), (9, 0.113)],
 [(2, 0.263), (12, 0.145)]]
```

We get the top two topics for each research paper because a paper or document can always be a mixture of multiple topics. Let's view the results for each paper in an easy-to-understand format.

```
results_df = pd.DataFrame()
results_df['Papers'] = range(1, len(new_papers)+1)
results_df['Dominant Topics'] = [[topic_num+1 for topic_num, wt in item]
                                     for item in topic_preds]
res = results_df.set_index(['Papers'])['Dominant Topics'].apply(pd.Series).
stack().reset_index(level=1, drop=True)
results_df = pd.DataFrame({'Dominant Topics': res.values}, index=res.index)
results_df['Contribution %'] = [topic_wt for topic_list in
                                    [[round(wt*100, 2)
                                          for topic_num, wt in item]
                                             for item in topic_preds]
                                    for topic_wt in topic_list]

results_df['Topic Desc'] = [topics_df.iloc[t-1]['Terms per Topic']
                               for t in results_df['Dominant Topics'].values]
results_df['Paper Desc'] = [new_papers[i-1][:200] for i in results_
df.index.values]
pd.set_option('display.max_colwidth', 300)
results_df
```



	Dominant Topics	Contribution %	Topic Desc	Paper Desc
Papers				
1	8	24.1	distribution, probability, prior, gaussian, variable, mixture, density, bayesian, estimate, approximation, log, likelihood, sample, component, expert, em, posterior, probabilistic, estimation, entropy	Cooperative Graphical Models\nJosip Djolonga\nDept. of Computer Science, ETH Zurich `\njosipd@inf.ethz.ch\nStefanie Jegelka\nCSAIL, MIT\nstefje@mit.edu\nSebastian Tschiatschek\nDept. of Computer Science, ETH
1	5	19.9	bound, theorem, class, probability, size, threshold, proof, polynomial, theory, complexity, loss, approximation, linear, assume, definition, defined, hypothesis, constant, define, bounded	Cooperative Graphical Models\nJosip Djolonga\nDept. of Computer Science, ETH Zurich `\njosipd@inf.ethz.ch\nStefanie Jegelka\nCSAIL, MIT\nstefje@mit.edu\nSebastian Tschiatschek\nDept. of Computer Science, ETH
2	14	29.3	control, action, state, policy, environment, controller, reinforcement_learning, task, optimal, robot, goal, step, reward, td, agent, adaptive, cost, reinforcement, trial, exploration	PAC Reinforcement Learning with Rich Observations\nAkshay Krishnamurthy\nUniversity of Massachusetts, Amherst\nAmherst, MA, 01003\nakshay@cs.umass.edu\nAlekh Agarwal\nMicrosoft Research\nNew York, NY 10011\na
2	5	24.8	bound, theorem, class, probability, size, threshold, proof, polynomial, theory, complexity, loss, approximation, linear, assume, definition, defined, hypothesis, constant, define, bounded	PAC Reinforcement Learning with Rich Observations\nAkshay Krishnamurthy\nUniversity of Massachusetts, Amherst\nAmherst, MA, 01003\nakshay@cs.umass.edu\nAlekh Agarwal\nMicrosoft Research\nNew York, NY 10011\na
3	13	23.8	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location	Automated scalable segmentation of neurons from\nmultispectral images\nUygar Sümbül\nGrossman Center for the Statistics of Mind\nand Dept. of Statistics, Columbia University\nDouglas Roossien Jr.\nUniversit
3	10	11.3	neuron, cell, spike, synaptic, activity, response, stimulus, firing, synapsis, et_al, effect, neural, neuronal, current, pattern, inhibitory, connection, brain, simulation, firing_rate	Automated scalable segmentation of neurons from\nmultispectral images\nUygar Sümbül\nGrossman Center for the Statistics of Mind\nand Dept. of Statistics, Columbia University\nDouglas Roossien Jr.\nUniversit
4	3	26.3	word, recognition, training, speech, character, context, hmm, letter, mlp, speaker, feature, frame, trained, speech_recognition, phoneme, experiment, hybrid, segmentation, vowel, level	Unsupervised Learning of Spoken Language with\nVisual Context\nDavid Harwath, Antonio Torralba, and James R. Glass\nComputer Science and Artificial Intelligence Laboratory\nMassachusetts Institute of Tech
4	13	14.5	image, object, feature, pixel, face, view, recognition, representation, shape, scale, part, visual, region, position, scene, surface, vision, frame, texture, location	Unsupervised Learning of Spoken Language with\nVisual Context\nDavid Harwath, Antonio Torralba, and James R. Glass\nComputer Science and Artificial Intelligence Laboratory\nMassachusetts Institute of Tech

Figure 6-13. *Predicting topics for new papers with our LDA model*

Looking at the generated topics for the new, previously unseen papers in Figure 6-13, I would say our model has done an excellent job!

Topic Models with Scikit-Learn

This section has been incorporated based on all the people who mentioned they use Scikit-Learn extensively and would love to use it for topic modeling too. The Scikit-Learn framework does offer a suite of techniques and methods for building topic models, although the flexibility in tuning or controlling these models is slightly limited as compared to Gensim. Nevertheless, we will build topic models using the following methods in this section:

- Latent Semantic Indexing (LSI)
- Latent Dirichlet Allocation (LDA)
- Non-negative Matrix Factorization (NMF)

We try to replicate the feature engineering process as much as possible based on what we did when we built topic models with Gensim. These models work on the same normalized and preprocessed corpus present in the norm_papers variable.

Text Representation with Feature Engineering

We represent our text data in the form of a Bag of Words model with uni-grams and bi-grams, similar to our analyses in the previous section.

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(min_df=20, max_df=0.6, ngram_range=(1,2),
                     token_pattern=None, tokenizer=lambda doc: doc,
                     preprocessor=lambda doc: doc)
cv_features = cv.fit_transform(norm_papers)
cv_features.shape

(1740, 14408)

# validating vocabulary size
vocabulary = np.array(cv.get_feature_names())
print('Total Vocabulary Size:', len(vocabulary))

Total Vocabulary Size: 14408
```

While the vocabulary is double what we had when we built models using Gensim, we have still removed unnecessary terms with the document frequency filters.

Latent Semantic Indexing

The first topic modeling technique we try is the LSI model based on SVD. Since we determined the optimal number of topics as 20 in the previous section, let’s use the name for the total topics we want to generate.

```
%%time
from sklearn.decomposition import TruncatedSVD

TOTAL_TOPICS = 20
```

```
lsi_model = TruncatedSVD(n_components=TOTAL_TOPICS, n_iter=500, random_
state=42)
document_topics = lsi_model.fit_transform(cv_features)
```

```
CPU times: user 15min 25s, sys: 1min 3s, total: 16min 28s
Wall time: 1min 1s
```

```
topic_terms = lsi_model.components_
topic_terms.shape
```

```
(20, 14408)
```

We can now generate the topics by reusing some of the code we implemented previously to display the topics and terms.

```
top_terms = 20
topic_key_term_idxs = np.argsort(-np.absolute(topic_terms), axis=1)[:,
:top_terms]
topic_keyterm_weights = np.array([topic_terms[row, columns]
                             for row, columns in list(zip(np.arange(TOTAL_
                             TOPICS), topic_key_term_idxs))])
topic_keyterms = vocabulary[topic_key_term_idxs]
topic_keyterms_weights = list(zip(topic_keyterms, topic_keyterm_weights))
for n in range(TOTAL_TOPICS):
    print('Topic #'+str(n+1)+':')
    print('='*50)
    d1 = []
    d2 = []
    terms, weights = topic_keyterms_weights[n]
    term_weights = sorted([(t, w) for t, w in zip(terms, weights)],
                          key=lambda row: -abs(row[1]))
    for term, wt in term_weights:
        if wt >= 0:
            d1.append((term, round(wt, 3)))
        else:
            d2.append((term, round(wt, 3)))

    print('Direction 1:', d1)
    print('-'*50)
    print('Direction 2:', d2)
    print('-'*50)
    print()
Topic #1:
==================================================
Direction 1: [('state', 0.221), ('neuron', 0.169), ('image', 0.138),
('cell', 0.13), ('layer', 0.13), ('feature', 0.127), ('probability',
0.121), ('hidden', 0.114), ('distribution', 0.105), ('rate', 0.098),
('signal', 0.095), ('task', 0.093), ('class', 0.092), ('noise', 0.09),
('net', 0.089), ('recognition', 0.089), ('representation', 0.088),
('field', 0.082), ('rule', 0.082), ('step', 0.08)]
--------------------------------------------------
Direction 2: []
--------------------------------------------------
...,

Topic #3:
==================================================
Direction 1: [('state', 0.574), ('neuron', 0.212), ('action', 0.187),
('policy', 0.149), ('control', 0.12), ('dynamic', 0.1), ('cell', 0.083),
('reinforcement', 0.081), ('optimal', 0.075), ('reinforcement learning',
0.068)]
--------------------------------------------------
Direction 2: [('image', -0.364), ('feature', -0.223), ('object', -0.144),
('recognition', -0.143), ('classifier', -0.111), ('class', -0.106), ('layer',
-0.092), ('classification', -0.085), ('face', -0.073), ('test', -0.069)]
--------------------------------------------------

Topic #4:
==================================================
Direction 1: [('image', 0.425), ('state', 0.326), ('object', 0.215),
('feature', 0.159), ('action', 0.147), ('visual', 0.143), ('control',
0.126), ('task', 0.111), ('policy', 0.103), ('recognition', 0.103),
('face', 0.092), ('representation', 0.086), ('motion', 0.086)]
--------------------------------------------------
Direction 2: [('neuron', -0.216), ('distribution', -0.166), ('class',
-0.112), ('bound', -0.109), ('probability', -0.108), ('spike', -0.104),
('variable', -0.087)]
--------------------------------------------------
...,

Topic #6:
==================================================
Direction 1: [('cell', 0.548), ('layer', 0.139), ('word', 0.124),
('hidden', 0.111), ('classifier', 0.097), ('direction', 0.09), ('head',
0.078), ('rule', 0.073), ('rat', 0.073), ('speech', 0.071)]
--------------------------------------------------
Direction 2: [('neuron', -0.416), ('image', -0.336), ('circuit', -0.126),
('noise', -0.124), ('chip', -0.121), ('analog', -0.099), ('object', -0.09),
('spike', -0.075), ('signal', -0.071), ('voltage', -0.069)]
--------------------------------------------------

...,

Topic #9:
==================================================
Direction 1: [('circuit', 0.244), ('control', 0.242), ('classifier',
0.229), ('chip', 0.167), ('node', 0.137), ('current', 0.132), ('analog',
0.13), ('voltage', 0.129), ('signal', 0.118), ('controller', 0.088)]
--------------------------------------------------
Direction 2: [('hidden', -0.27), ('neuron', -0.247), ('state',
-0.175), ('distribution', -0.158), ('hidden unit', -0.143), ('layer',
-0.125), ('object', -0.115), ('probability', -0.108), ('image', -0.1),
('representation', -0.098)]
--------------------------------------------------

Topic #10:
==================================================
Direction 1: [('circuit', 0.245), ('cell', 0.225), ('node', 0.211),
('state', 0.183), ('image', 0.166), ('chip', 0.163), ('analog', 0.147),
('layer', 0.144), ('net', 0.12), ('voltage', 0.115)]
--------------------------------------------------
Direction 2: [('task', -0.201), ('rule', -0.193), ('spike', -0.166),
('feature', -0.165), ('control', -0.157), ('neuron', -0.144), ('rate',
-0.134), ('stimulus', -0.116), ('classifier', -0.116), ('action', -0.112)]
--------------------------------------------------
...,

Topic #18:
==================================================
Direction 1: [('object', 0.419), ('signal', 0.26), ('layer', 0.258),
('rule', 0.209), ('feature', 0.164), ('view', 0.162), ('net', 0.113),
('noise', 0.112), ('bound', 0.105), ('speech', 0.1)]
--------------------------------------------------
Direction 2: [('memory', -0.18), ('task', -0.161), ('representation',
-0.14), ('hidden', -0.137), ('image', -0.135), ('hidden unit', -0.121),
('tree', -0.117), ('structure', -0.094), ('test', -0.093), ('word', -0.092)]
--------------------------------------------------

Topic #19:
==================================================
Direction 1: [('class', 0.287), ('memory', 0.275), ('classifier', 0.144),
('response', 0.139), ('sequence', 0.112), ('component', 0.11), ('stimulus',
0.101), ('region', 0.092), ('bound', 0.088)]
--------------------------------------------------
Direction 2: [('node', -0.292), ('feature', -0.244), ('field', -0.202),
('rate', -0.152), ('word', -0.146), ('spike', -0.139), ('map', -0.132),
('character', -0.127), ('policy', -0.108), ('tree', -0.092), ('noise', -0.088)]
--------------------------------------------------

Topic #20:
==================================================
Direction 1: [('map', 0.222), ('control', 0.2), ('region', 0.181), ('ii',
0.145), ('feature', 0.132), ('image', 0.122), ('bound', 0.11), ('orientation',
0.109), ('rule', 0.109), ('threshold', 0.094), ('class', 0.092)]
```

```
--------------------------------------------------
Direction 2: [('object', -0.31), ('motion', -0.252), ('direction', -0.229),
('memory', -0.223), ('classifier', -0.193), ('view', -0.136), ('matrix',
-0.13), ('rate', -0.121), ('distance', -0.11)]
--------------------------------------------------
```

Of course, we don't show all the topics in the preceding output due to space constraints, but you get the general idea and you can view all the topics in the notebook if needed. Similar to the previous section, we can also extract key topics for specific research papers.

```
dt_df = pd.DataFrame(np.round(document_topics, 3),
                    columns=['T'+str(i) for i in range(1, TOTAL_TOPICS+1)])
document_numbers = [13, 250, 500]

for document_number in document_numbers:
    top_topics = list(dt_df.columns[np.argsort(-
                           np.absolute(dt_df.iloc[document_number].
                           values))[:3]])
    print('Document #'+str(document_number)+':')
    print('Dominant Topics (top 3):', top_topics)
    print('Paper Summary:')
    print(papers[document_number][:500])
    print()

Document #13:
Dominant Topics (top 3): ['T1', 'T6', 'T4']
Paper Summary:
Stochastic Learning Networks and their Electronic Implementation
Joshua Alspector*, Robert B. Allen, Victor Hut, and Srinagesh Satyanarayana
Bell Communications Research, Morristown, NJ 07960
ABSTRACT
We describe a family of learning algorithms that operate on a recurrent,
symmetrically connected, neuromorphic network that, like the Boltzmann
machine

Document #250:
Dominant Topics (top 3): ['T3', 'T18', 'T4']
```

```
Paper Summary:
266 Zemel, Mozer and Hinton
TRAFFIC: Recognizing Objects Using
Hierarchical Reference Frame Transformations
Richard S. Zemel
ABSTRACT
We describe a model that can recognize two-dimensional shapes in
an unsegmented image, independent of their orie

Document #500:
Dominant Topics (top 3): ['T9', 'T1', 'T10']
Paper Summary:
Constrained Optimization Applied to the
Parameter Setting Problem for Analog Circuits
David Kirk, Kurt Fleischer, Lloyd Watts, Alan Bart
Abstract
We use constrained optimization to select operating parameters for two
circuits: a simple 3-transistor square root circuit, and an analog VLSI
artificial cochlea.
```

If you check out the terms in the topics we obtained in the preceding output, they actually make sense!

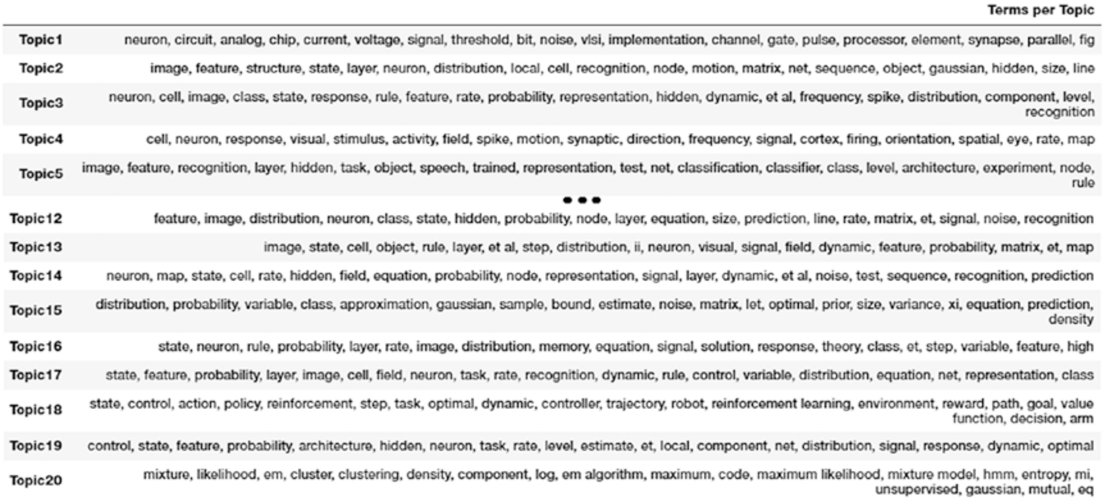
Latent Dirichlet Allocation

Even Scikit-Learn has included an LDA-based topic model implementation in their library and the following snippet uses it to build an LDA topic model.

```
%%time
from sklearn.decomposition import LatentDirichletAllocation

lda_model = LatentDirichletAllocation(n_components =TOTAL_TOPICS,
max_iter=500, max_doc_update_iter=50, learning_method='online',
batch_size=1740, learning_offset=50., random_state=42, n_jobs=16)
document_topics = lda_model.fit_transform(cv_features)

CPU times: user 13min 14s, sys: 1min 41s, total: 14min 56s
Wall time: 55min 32s
```

We can then obtain the topic-term matrix and build a dataframe from it to showcase the topics and terms in an easy-to-interpret format.

```
topic_terms = lda_model.components_
topic_key_term_idxs = np.argsort(-np.absolute(topic_terms), axis=1)[:,
:top_terms]
topic_keyterms = vocabulary[topic_key_term_idxs]
topics = [', '.join(topic) for topic in topic_keyterms]
pd.set_option('display.max_colwidth', -1)
topics_df = pd.DataFrame(topics,
                         columns = ['Terms per Topic'],
                         index=['Topic'+str(t) for t in range(1, TOTAL_
                         TOPICS+1)])
topics_df
```



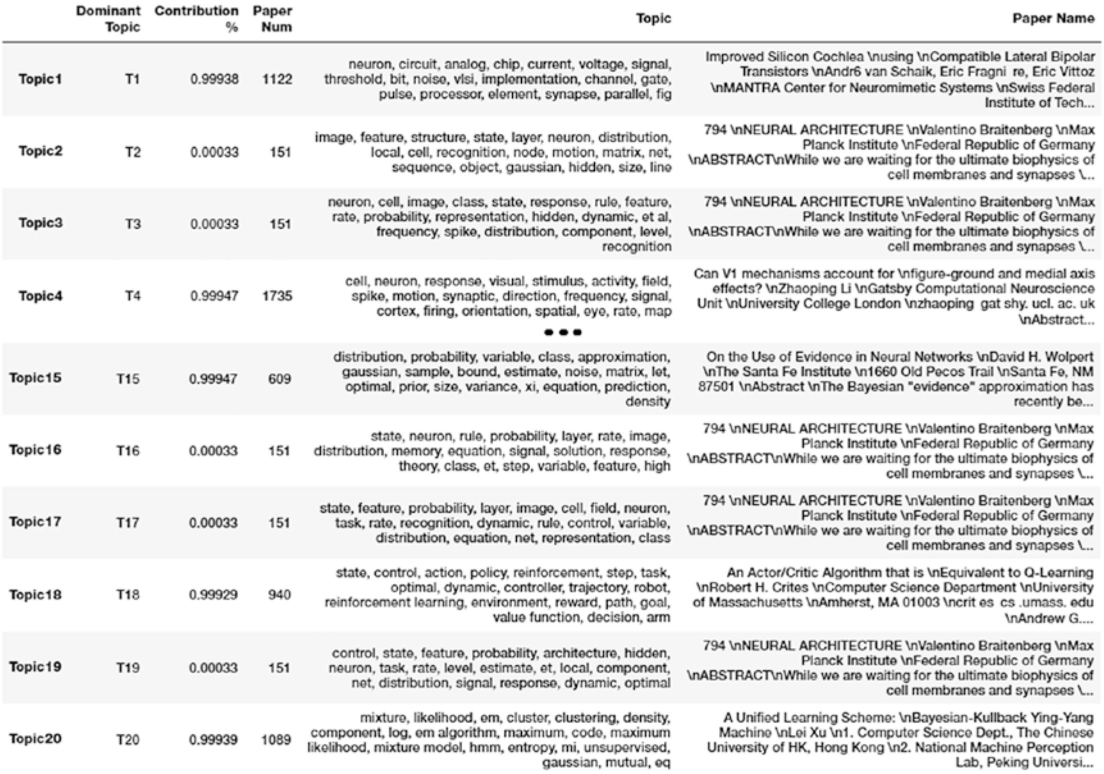
	Terms per Topic
Topic1	neuron, circuit, analog, chip, current, voltage, signal, threshold, bit, noise, vlsi, implementation, channel, gate, pulse, processor, element, synapse, parallel, fig
Topic2	image, feature, structure, state, layer, neuron, distribution, local, cell, recognition, node, motion, matrix, net, sequence, object, gaussian, hidden, size, line
Topic3	neuron, cell, image, class, state, response, rule, feature, rate, probability, representation, hidden, dynamic, et al, frequency, spike, distribution, component, level, recognition
Topic4	cell, neuron, response, visual, stimulus, activity, field, spike, motion, synaptic, direction, frequency, signal, cortex, firing, orientation, spatial, eye, rate, map
Topic5	image, feature, recognition, layer, hidden, task, object, speech, trained, representation, test, net, classification, classifier, class, level, architecture, experiment, node, rule
	•••
Topic12	feature, image, distribution, neuron, class, state, hidden, probability, node, layer, equation, size, prediction, line, rate, matrix, et, signal, noise, recognition
Topic13	image, state, cell, object, rule, layer, et al, step, distribution, ii, neuron, visual, signal, field, dynamic, feature, probability, matrix, et, map
Topic14	neuron, map, state, cell, rate, hidden, field, equation, probability, node, representation, signal, layer, dynamic, et al, noise, test, sequence, recognition, prediction
Topic15	distribution, probability, variable, class, approximation, gaussian, sample, bound, estimate, noise, matrix, let, optimal, prior, size, variance, xi, equation, prediction, density
Topic16	state, neuron, rule, probability, layer, rate, image, distribution, memory, equation, signal, solution, response, theory, class, et, step, variable, feature, high
Topic17	state, feature, probability, layer, image, cell, field, neuron, task, rate, recognition, dynamic, rule, control, variable, distribution, equation, net, representation, class
Topic18	state, control, action, policy, reinforcement, step, task, optimal, dynamic, controller, trajectory, robot, reinforcement learning, environment, reward, path, goal, value function, decision, arm
Topic19	control, state, feature, probability, architecture, hidden, neuron, task, rate, level, estimate, et, local, component, net, distribution, signal, response, dynamic, optimal
Topic20	mixture, likelihood, em, cluster, clustering, density, component, log, em algorithm, maximum, code, maximum likelihood, mixture model, hmm, entropy, mi, unsupervised, gaussian, mutual, eq

Figure 6-14. *Generated topics from our LDA model*

Based on the topics depicted in Figure 6-14, we can see some repetition in similar themes among the topics, which might be an indication that this model is not as good as our MALLET LDA model. We can now view the research papers having the maximum contribution of each of the 20 topics, similar to our analyses in the previous sections.

```
dt_df = pd.DataFrame(document_topics,
                     columns=['T'+str(i) for i in range(1, TOTAL_TOPICS+1)])
```

```
pd.options.display.float_format = '{:,.5f}'.format
pd.set_option('display.max_colwidth', 200)

max_contrib_topics = dt_df.max(axis=0)
dominant_topics = max_contrib_topics.index
contrib_perc = max_contrib_topics.values
document_numbers = [dt_df[dt_df[t] == max_contrib_topics.loc[t]].index[0]
                       for t in dominant_topics]
documents = [papers[i] for i in document_numbers]

results_df = pd.DataFrame({'Dominant Topic': dominant_topics, 'Contribution
                          %': contrib_perc,
                          'Paper Num': document_numbers, 'Topic': topics_
                          df['Terms per Topic'],
                          'Paper Name': documents})
results_df
```



	Dominant Topic	Contribution %	Paper Num	Topic	Paper Name
Topic1	T1	0.99938	1122	neuron, circuit, analog, chip, current, voltage, signal, threshold, bit, noise, vlsi, implementation, channel, gate, pulse, processor, element, synapse, parallel, fig	Improved Silicon Cochlea \nusing \nCompatible Lateral Bipolar Transistors \nAndr6 van Schaik, Eric Fragni re, Eric Vittoz \nMANTRA Center for Neuromimetic Systems \nSwiss Federal Institute of Tech...
Topic2	T2	0.00033	151	image, feature, structure, state, layer, neuron, distribution, local, cell, recognition, node, motion, matrix, net, sequence, object, gaussian, hidden, size, line	794 \nNEURAL ARCHITECTURE \nValentino Braitenberg \nMax Planck Institute \nFederal Republic of Germany \nABSTRACT\nWhile we are waiting for the ultimate biophysics of cell membranes and synapses \...
Topic3	T3	0.00033	151	neuron, cell, image, class, state, response, rule, feature, rate, probability, representation, hidden, dynamic, et al, frequency, spike, distribution, component, level, recognition	794 \nNEURAL ARCHITECTURE \nValentino Braitenberg \nMax Planck Institute \nFederal Republic of Germany \nABSTRACT\nWhile we are waiting for the ultimate biophysics of cell membranes and synapses \...
Topic4	T4	0.99947	1735	cell, neuron, response, visual, stimulus, activity, field, spike, motion, synaptic, direction, frequency, signal, cortex, firing, orientation, spatial, eye, rate, map	Can V1 mechanisms account for \nfigure-ground and medial axis effects? \nZhaoping Li \nGatsby Computational Neuroscience Unit \nUniversity College London \nzhaoping gat shy. ucl. ac. uk \nAbstract...
				•••	
Topic15	T15	0.99947	609	distribution, probability, variable, class, approximation, gaussian, sample, bound, estimate, noise, matrix, let, optimal, prior, size, variance, xi, equation, prediction, density	On the Use of Evidence in Neural Networks \nDavid H. Wolpert \nThe Santa Fe Institute \n1660 Old Pecos Trail \nSanta Fe, NM 87501 \nAbstract \nThe Bayesian "evidence" approximation has recently be...
Topic16	T16	0.00033	151	state, neuron, rule, probability, layer, rate, image, distribution, memory, equation, signal, solution, response, theory, class, et, step, variable, feature, high	794 \nNEURAL ARCHITECTURE \nValentino Braitenberg \nMax Planck Institute \nFederal Republic of Germany \nABSTRACT\nWhile we are waiting for the ultimate biophysics of cell membranes and synapses \...
Topic17	T17	0.00033	151	state, feature, probability, layer, image, cell, field, neuron, task, rate, recognition, dynamic, rule, control, variable, distribution, equation, net, representation, class	794 \nNEURAL ARCHITECTURE \nValentino Braitenberg \nMax Planck Institute \nFederal Republic of Germany \nABSTRACT\nWhile we are waiting for the ultimate biophysics of cell membranes and synapses \...
Topic18	T18	0.99929	940	state, control, action, policy, reinforcement, step, task, optimal, dynamic, controller, trajectory, robot, reinforcement learning, environment, reward, path, goal, value function, decision, arm	An Actor/Critic Algorithm that is \nEquivalent to Q-Learning \nRobert H. Crites \nComputer Science Department \nUniversity of Massachusetts \nAmherst, MA 01003 \ncrit es cs .umass. edu \nAndrew G....
Topic19	T19	0.00033	151	control, state, feature, probability, architecture, hidden, neuron, task, rate, level, estimate, et, local, component, net, distribution, signal, response, dynamic, optimal	794 \nNEURAL ARCHITECTURE \nValentino Braitenberg \nMax Planck Institute \nFederal Republic of Germany \nABSTRACT\nWhile we are waiting for the ultimate biophysics of cell membranes and synapses \...
Topic20	T20	0.99939	1089	mixture, likelihood, em, cluster, clustering, density, component, log, em algorithm, maximum, code, maximum likelihood, mixture model, hmm, entropy, mi, unsupervised, gaussian, mutual, eq	A Unified Learning Scheme: \nBayesian-Kullback Ying-Yang Machine \nLei Xu \n1. Computer Science Dept., The Chinese University of HK, Hong Kong \n2. National Machine Perception Lab, Peking Universi...

Figure 6-15. *Viewing each topic and corresponding paper with its maximum contribution*

Based on the output depicted in Figure 6-15, we can see that some topics have a very poor representation of almost 0% in the corpus and so we see the same paper (Paper #151) being selected as the more relevant paper for these topics. The topics with a good contribution (almost 100% dominance) showcase papers that are closely correlated with the theme conveyed by the corresponding topic, including reinforcement learning, Bayesian and Gaussian mixture models, neural models on VLSI, and transistors.

Non-Negative Matrix Factorization

The last technique we look at is non-negative matrix factorization (NMF), which is another matrix decomposition technique similar to SVD but operates on non-negative matrices and works well for multivariate data. Given a non-negative matrix *V*, the objective of NMF is to find two non-negative matrix factors, **W** and **H**, such that when they are multiplied, they can approximately reconstruct **V**. Mathematically this is represented as follows:

$$V \approx WH$$

such that all three matrices are non-negative. To get to this approximation, we usually use a cost function like the Euclidean distance or L2 norm between two matrices or the Frobenius norm, which is a slight modification of the L2 norm. This can be represented as follows:

$$\arg\min_{W,H} \frac{1}{2} \| V - WH \|^2$$

where we have our three non-negative matrices—**V**, **W**, and **H**—and this can be further simplified as follows:

$$\frac{1}{2} \sum_{i,j} \left(V_{ij} - WH_{ij} \right)^2$$

This implementation is available in the NMF class in the Scikit-Learn decomposition module, which we use in the section.

We can build an NMF based topic model using the following snippet on our toy corpus, which gives us the feature names and their weights just like in LDA.

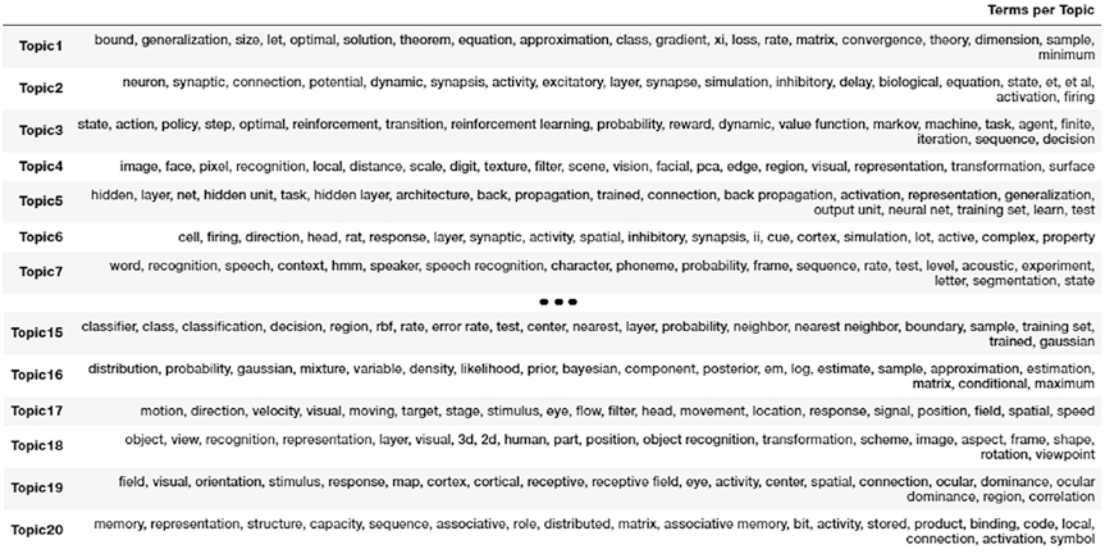
```
%%time
from sklearn.decomposition import NMF
```

```
nmf_model = NMF(n_components=TOTAL_TOPICS, solver='cd', max_iter=500,
                random_state=42, alpha=.1, l1_ratio=.85)
document_topics = nmf_model.fit_transform(cv_features)
```

```
CPU times: user 11min 39s, sys: 47.5 s, total: 12min 26s
Wall time: 46.7 s
```

Now that we have our model trained, we can look at the generated topics using the following code.

```
topic_terms = nmf_model.components_
topic_key_term_idxs = np.argsort(-np.absolute(topic_terms), axis=1)[:,
:top_terms]
topic_keyterms = vocabulary[topic_key_term_idxs]
topics = [', '.join(topic) for topic in topic_keyterms]
pd.set_option('display.max_colwidth', -1)
topics_df = pd.DataFrame(topics,
                         columns = ['Terms per Topic'],
                         index=['Topic'+str(t) for t in range(1, TOTAL_
                         TOPICS+1)])
topics_df
```

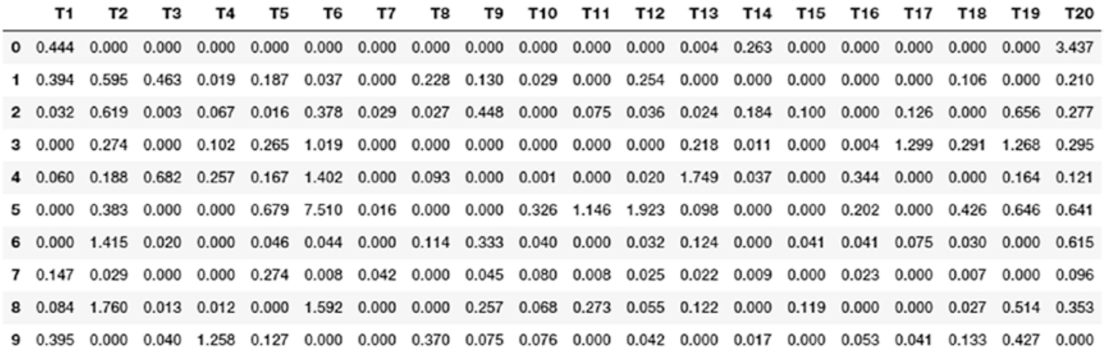


	Terms per Topic
Topic1	bound, generalization, size, let, optimal, solution, theorem, equation, approximation, class, gradient, xi, loss, rate, matrix, convergence, theory, dimension, sample, minimum
Topic2	neuron, synaptic, connection, potential, dynamic, synapsis, activity, excitatory, layer, synapse, simulation, inhibitory, delay, biological, equation, state, et, et al, activation, firing
Topic3	state, action, policy, step, optimal, reinforcement, transition, reinforcement learning, probability, reward, dynamic, value function, markov, machine, task, agent, finite, iteration, sequence, decision
Topic4	image, face, pixel, recognition, local, distance, scale, digit, texture, filter, scene, vision, facial, pca, edge, region, visual, representation, transformation, surface
Topic5	hidden, layer, net, hidden unit, task, hidden layer, architecture, back, propagation, trained, connection, back propagation, activation, representation, generalization, output unit, neural net, training set, learn, test
Topic6	cell, firing, direction, head, rat, response, layer, synaptic, activity, spatial, inhibitory, synapsis, ii, cue, cortex, simulation, lot, active, complex, property
Topic7	word, recognition, speech, context, hmm, speaker, speech recognition, character, phoneme, probability, frame, sequence, rate, test, level, acoustic, experiment, letter, segmentation, state
	•••
Topic15	classifier, class, classification, decision, region, rbf, rate, error rate, test, center, nearest, layer, probability, neighbor, nearest neighbor, boundary, sample, training set, trained, gaussian
Topic16	distribution, probability, gaussian, mixture, variable, density, likelihood, prior, bayesian, component, posterior, em, log, estimate, sample, approximation, estimation, matrix, conditional, maximum
Topic17	motion, direction, velocity, visual, moving, target, stage, stimulus, eye, flow, filter, head, movement, location, response, signal, position, field, spatial, speed
Topic18	object, view, recognition, representation, layer, visual, 3d, 2d, human, part, position, object recognition, transformation, scheme, image, aspect, frame, shape, rotation, viewpoint
Topic19	field, visual, orientation, stimulus, response, map, cortex, cortical, receptive, receptive field, eye, activity, center, spatial, connection, ocular, dominance, ocular dominance, region, correlation
Topic20	memory, representation, structure, capacity, sequence, associative, role, distributed, matrix, associative memory, bit, activity, stored, product, binding, code, local, connection, activation, symbol

Figure 6-16. *Generated topics from our NMF model*

Based on the topics depicted in Figure 6-16, there are no major repetitions of topics and each topic talks about a clear and distinct theme. The results from the NMF topic model are definitely better than what we obtained from LDA in Scikit-Learn. We can determine the dominance of topics in each research paper but, in case of NMF these are determined by absolute scores and not percentages, as depicted in the following output. See Figure 6-17.

```
pd.options.display.float_format = '{:,.3f}'.format
dt_df = pd.DataFrame(document_topics,
                     columns=['T'+str(i) for i in range(1, TOTAL_TOPICS+1)])
dt_df.head(10)
```



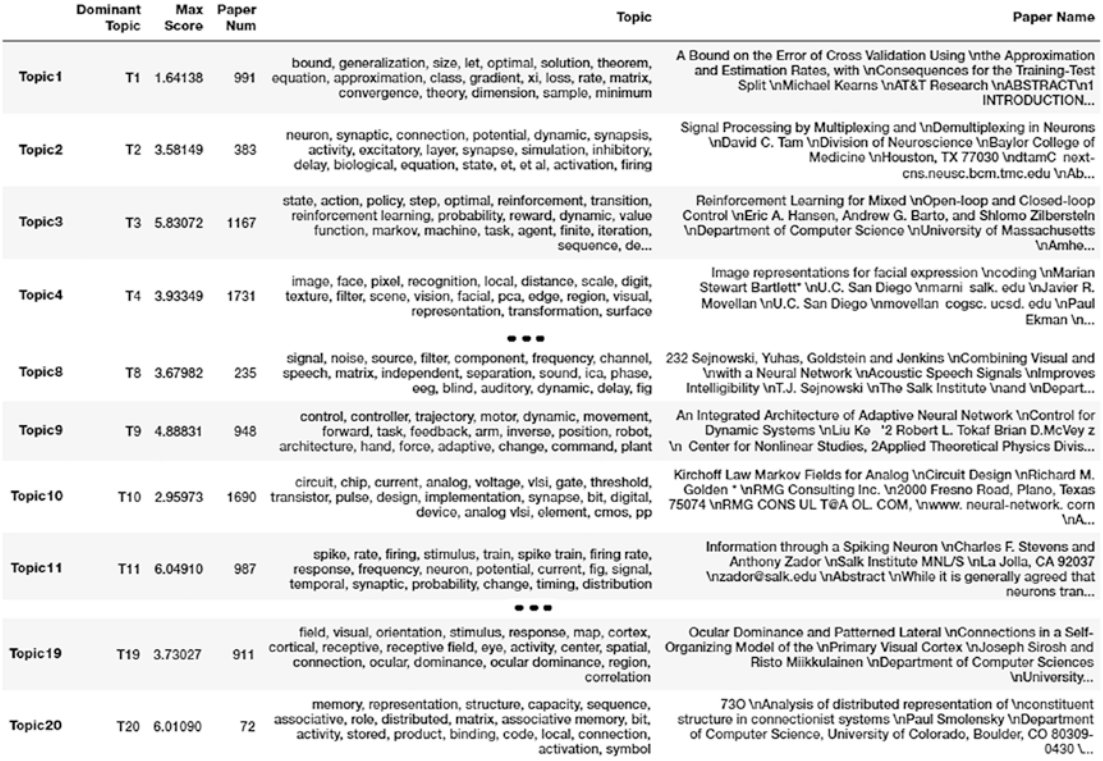
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20
0	0.444	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.263	0.000	0.000	0.000	0.000	0.000	3.437
1	0.394	0.595	0.463	0.019	0.187	0.037	0.000	0.228	0.130	0.029	0.000	0.254	0.000	0.000	0.000	0.000	0.000	0.106	0.000	0.210
2	0.032	0.619	0.003	0.067	0.016	0.378	0.029	0.027	0.448	0.000	0.075	0.036	0.024	0.184	0.100	0.000	0.126	0.000	0.656	0.277
3	0.000	0.274	0.000	0.102	0.265	1.019	0.000	0.000	0.000	0.000	0.000	0.000	0.218	0.011	0.000	0.004	1.299	0.291	1.268	0.295
4	0.060	0.188	0.682	0.257	0.167	1.402	0.000	0.093	0.000	0.001	0.000	0.020	1.749	0.037	0.000	0.344	0.000	0.000	0.164	0.121
5	0.000	0.383	0.000	0.000	0.679	7.510	0.016	0.000	0.000	0.326	1.146	1.923	0.098	0.000	0.000	0.202	0.000	0.426	0.646	0.641
6	0.000	1.415	0.020	0.000	0.046	0.044	0.000	0.114	0.333	0.040	0.000	0.032	0.124	0.000	0.041	0.041	0.075	0.030	0.000	0.615
7	0.147	0.029	0.000	0.000	0.274	0.008	0.042	0.000	0.045	0.080	0.008	0.025	0.022	0.009	0.000	0.023	0.000	0.007	0.000	0.096
8	0.084	1.760	0.013	0.012	0.000	1.592	0.000	0.000	0.257	0.068	0.273	0.055	0.122	0.000	0.119	0.000	0.000	0.027	0.514	0.353
9	0.395	0.000	0.040	1.258	0.127	0.000	0.000	0.370	0.075	0.076	0.000	0.042	0.000	0.017	0.000	0.053	0.041	0.133	0.427	0.000

Figure 6-17. *Viewing topic dominance per document using the document-topic matrix*

Leveraging the document-topic matrix, we can determine the most relevant paper for each topic based on the topic dominance scores by using the following code.

```
pd.options.display.float_format = '{:,.5f}'.format
pd.set_option('display.max_colwidth', 200)

max_score_topics = dt_df.max(axis=0)
dominant_topics = max_score_topics.index
term_score = max_score_topics.values
document_numbers = [dt_df[dt_df[t] == max_score_topics.loc[t]].index[0]
                         for t in dominant_topics]
documents = [papers[i] for i in document_numbers]
```

```
results_df = pd.DataFrame({'Dominant Topic': dominant_topics, 'Max Score':
                           term_score,
                           'Paper Num': document_numbers, 'Topic': topics_
                           df['Terms per Topic'],
                           'Paper Name': documents})
results_df
```



	Dominant Topic	Max Score	Paper Num	Topic	Paper Name
Topic1	T1	1.64138	991	bound, generalization, size, let, optimal, solution, theorem, equation, approximation, class, gradient, xi, loss, rate, matrix, convergence, theory, dimension, sample, minimum	A Bound on the Error of Cross Validation Using \nthe Approximation and Estimation Rates, with \nConsequences for the Training-Test Split \nMichael Kearns \nAT&T Research \nABSTRACT\n1 INTRODUCTION...
Topic2	T2	3.58149	383	neuron, synaptic, connection, potential, dynamic, synapsis, activity, excitatory, layer, synapse, simulation, inhibitory, delay, biological, equation, state, et, et al, activation, firing	Signal Processing by Multiplexing and \nDemultiplexing in Neurons \nDavid C. Tam \nDivision of Neuroscience \nBaylor College of Medicine \nHouston, TX 77030 \ndtamC next-cns.neusc.bcm.tmc.edu \nAb...
Topic3	T3	5.83072	1167	state, action, policy, step, optimal, reinforcement, transition, reinforcement learning, probability, reward, dynamic, value function, markov, machine, task, agent, finite, iteration, sequence, de...	Reinforcement Learning for Mixed \nOpen-loop and Closed-loop Control \nEric A. Hansen, Andrew G. Barto, and Shlomo Zilbersteln \nDepartment of Computer Science \nUniversity of Massachusetts \nAmhe...
Topic4	T4	3.93349	1731	image, face, pixel, recognition, local, distance, scale, digit, texture, filter, scene, vision, facial, pca, edge, region, visual, representation, transformation, surface	Image representations for facial expression \ncoding \nMarian Stewart Bartlett* \nU.C. San Diego \nmarni salk. edu \nJavier R. Movellan \nU.C. San Diego \nmovellan cogsc. ucsd. edu \nPaul Ekman \n...
				•••	
Topic8	T8	3.67982	235	signal, noise, source, filter, component, frequency, channel, speech, matrix, independent, separation, sound, ica, phase, eeg, blind, auditory, dynamic, delay, fig	232 Sejnowski, Yuhas, Goldstein and Jenkins \nCombining Visual and \nwith a Neural Network \nAcoustic Speech Signals \nImproves Intelligibility \nT.J. Sejnowski \nThe Salk Institute \nand \nDepart...
Topic9	T9	4.88831	948	control, controller, trajectory, motor, dynamic, movement, forward, task, feedback, arm, inverse, position, robot, architecture, hand, force, adaptive, change, command, plant	An Integrated Architecture of Adaptive Neural Network \nControl for Dynamic Systems \nLiu Ke '2 Robert L. Tokaf Brian D.McVey z \n Center for Nonlinear Studies, 2Applied Theoretical Physics Divis...
Topic10	T10	2.95973	1690	circuit, chip, current, analog, voltage, vlsi, gate, threshold, transistor, pulse, design, implementation, synapse, bit, digital, device, analog vlsi, element, cmos, pp	Kirchoff Law Markov Fields for Analog \nCircuit Design \nRichard M. Golden * \nRMG Consulting Inc. \n2000 Fresno Road, Plano, Texas 75074 \nRMG CONS UL T@A OL. COM, \nwww. neural-network. corn \nA...
Topic11	T11	6.04910	987	spike, rate, firing, stimulus, train, spike train, firing rate, response, frequency, neuron, potential, current, fig, signal, temporal, synaptic, probability, change, timing, distribution	Information through a Spiking Neuron \nCharles F. Stevens and Anthony Zador \nSalk Institute MNL/S \nLa Jolla, CA 92037 \nzador@salk.edu \nAbstract \nWhile it is generally agreed that neurons tran...
				•••	
Topic19	T19	3.73027	911	field, visual, orientation, stimulus, response, map, cortex, cortical, receptive, receptive field, eye, activity, center, spatial, connection, ocular, dominance, ocular dominance, region, correlation	Ocular Dominance and Patterned Lateral \nConnections in a Self-Organizing Model of the \nPrimary Visual Cortex \nJoseph Sirosh and Risto Miikkulainen \nDepartment of Computer Sciences \nUniversity...
Topic20	T20	6.01090	72	memory, representation, structure, capacity, sequence, associative, role, distributed, matrix, associative memory, bit, activity, stored, product, binding, code, local, connection, activation, symbol	73O \nAnalysis of distributed representation of \nconstituent structure in connectionist systems \nPaul Smolensky \nDepartment of Computer Science, University of Colorado, Boulder, CO 80309-0430 \...

Figure 6-18. *Viewing each topic and corresponding paper with its maximum contribution*

The outputs depicted in Figure 6-18 clearly show that the NMF model is much better than the LDA model, with each topic being strongly correlated as the central theme of the research paper where it has maximum dominance. What we have observed is that non-negative matrix factorization works the best even with small corpora, with few documents compared to the other methods. But again, this depends on the type of data you are dealing with.

Predicting Topics for New Research Papers

We now predict topics for the four research papers from the NIPS 16 conference, similar to what we did with the Gensim topic models. Start by loading the papers if you don't have them loaded already.

```
import glob
# papers manually downloaded from NIPS 16
# https://papers.nips.cc/book/advances-in-neural-information-processing-
systems-29-2016

new_paper_files = glob.glob('nips16*.txt')
new_papers = []
for fn in new_paper_files:
    with open(fn, encoding='utf-8', errors='ignore', mode='r+') as f:
        data = f.read()
        new_papers.append(data)

print('Total New Papers:', len(new_papers))
```

```
Total New Papers: 4
```

The next step in the pipeline is to preprocess these documents and extract features using the same sequence of steps we followed when building the topic models.

```
norm_new_papers = normalize_corpus(new_papers)
cv_new_features = cv.transform(norm_new_papers)
cv_new_features.shape
```

```
(4, 14408)
```

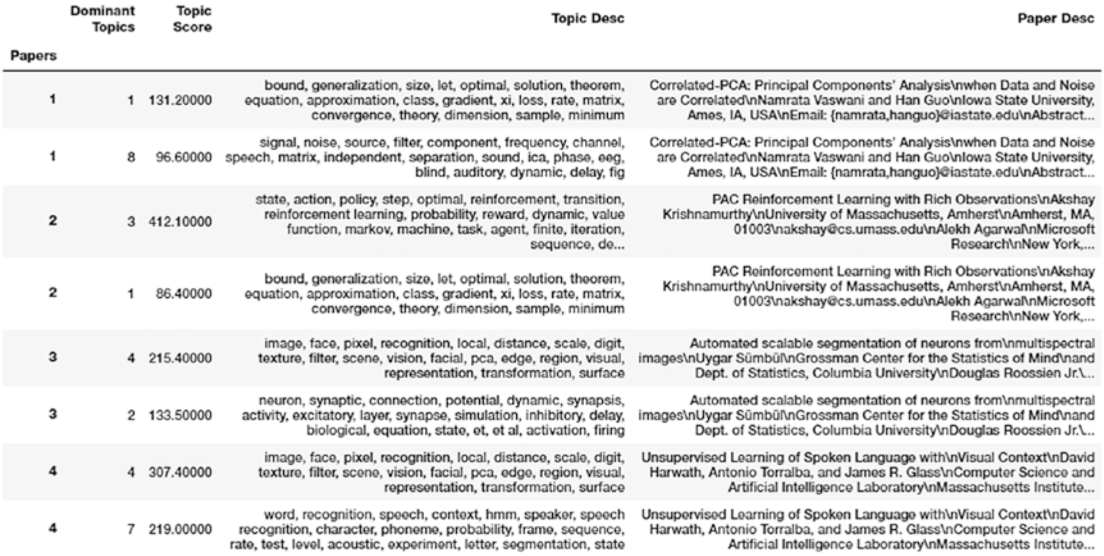
We can now use our NMF topic model to predict the topics for these new research papers using the following code (we predict the top two topics for each paper).

```
topic_predictions = nmf_model.transform(cv_new_features)
best_topics = [[(topic, round(sc, 3))
                    for topic, sc in sorted(enumerate(topic_predictions[i]),
                                            key=lambda row: -row[1])[:2]]
                        for i in range(len(topic_predictions))]
best_topics
```

```
[[(0, 1.312), (7, 0.966)],
 [(2, 4.121), (0, 0.864)],
 [(3, 2.154), (1, 1.335)],
 [(3, 3.074), (6, 2.19)]]
```

Remember that we don't get proportion of dominance of each topic here, like with the LDA model, but we get absolute scores. Let's view the results in an easy-to-understand format.

```
results_df = pd.DataFrame()
results_df['Papers'] = range(1, len(new_papers)+1)
results_df['Dominant Topics'] = [[topic_num+1 for topic_num, sc in item]
                                     for item in best_topics]
res = results_df.set_index(['Papers'])['Dominant Topics'].apply(pd.Series).
stack().reset_index(level=1, drop=True)
results_df = pd.DataFrame({'Dominant Topics': res.values}, index=res.index)
results_df['Topic Score'] = [topic_sc for topic_list in
                                        [[round(sc*100, 2)
                                              for topic_num, sc in item]
                                                 for item in best_topics]
                                    for topic_sc in topic_list]

results_df['Topic Desc'] = [topics_df.iloc[t-1]['Terms per Topic']
                            for t in results_df['Dominant Topics'].values]
results_df['Paper Desc'] = [new_papers[i-1][:200] for i in results_
df.index.values]
results_df
```



	Dominant Topics	Topic Score	Topic Desc	Paper Desc
Papers				
1	1	131.20000	bound, generalization, size, let, optimal, solution, theorem, equation, approximation, class, gradient, xi, loss, rate, matrix, convergence, theory, dimension, sample, minimum	Correlated-PCA: Principal Components' Analysis\nwhen Data and Noise are Correlated\nNamrata Vaswani and Han Guo\nIowa State University, Ames, IA, USA\nEmail: {namrata,hanguo}@iastate.edu\nAbstract...
1	8	96.60000	signal, noise, source, filter, component, frequency, channel, speech, matrix, independent, separation, sound, ica, phase, eeg, blind, auditory, dynamic, delay, fig	Correlated-PCA: Principal Components' Analysis\nwhen Data and Noise are Correlated\nNamrata Vaswani and Han Guo\nIowa State University, Ames, IA, USA\nEmail: {namrata,hanguo}@iastate.edu\nAbstract...
2	3	412.10000	state, action, policy, step, optimal, reinforcement, transition, reinforcement learning, probability, reward, dynamic, value function, markov, machine, task, agent, finite, iteration, sequence, de...	PAC Reinforcement Learning with Rich Observations\nAkshay Krishnamurthy\nUniversity of Massachusetts, Amherst\nAmherst, MA, 01003\nakshay@cs.umass.edu\nAlekh Agarwal\nMicrosoft Research\nNew York,...
2	1	86.40000	bound, generalization, size, let, optimal, solution, theorem, equation, approximation, class, gradient, xi, loss, rate, matrix, convergence, theory, dimension, sample, minimum	PAC Reinforcement Learning with Rich Observations\nAkshay Krishnamurthy\nUniversity of Massachusetts, Amherst\nAmherst, MA, 01003\nakshay@cs.umass.edu\nAlekh Agarwal\nMicrosoft Research\nNew York,...
3	4	215.40000	image, face, pixel, recognition, local, distance, scale, digit, texture, filter, scene, vision, facial, pca, edge, region, visual, representation, transformation, surface	Automated scalable segmentation of neurons from\nmultispectral images\nUygar Sümbül\nGrossman Center for the Statistics of Mind\nand Dept. of Statistics, Columbia University\nDouglas Roossien Jr.\...
3	2	133.50000	neuron, synaptic, connection, potential, dynamic, synapsis, activity, excitatory, layer, synapse, simulation, inhibitory, delay, biological, equation, state, et, et al, activation, firing	Automated scalable segmentation of neurons from\nmultispectral images\nUygar Sümbül\nGrossman Center for the Statistics of Mind\nand Dept. of Statistics, Columbia University\nDouglas Roossien Jr.\...
4	4	307.40000	image, face, pixel, recognition, local, distance, scale, digit, texture, filter, scene, vision, facial, pca, edge, region, visual, representation, transformation, surface	Unsupervised Learning of Spoken Language with\nVisual Context\nDavid Harwath, Antonio Torralba, and James R. Glass\nComputer Science and Artificial Intelligence Laboratory\nMassachusetts Institute...
4	7	219.00000	word, recognition, speech, context, hmm, speaker, speech recognition, character, phoneme, probability, frame, sequence, rate, test, level, acoustic, experiment, letter, segmentation, state	Unsupervised Learning of Spoken Language with\nVisual Context\nDavid Harwath, Antonio Torralba, and James R. Glass\nComputer Science and Artificial Intelligence Laboratory\nMassachusetts Institute...

Figure 6-19. *Predicting topics for new papers with our NMF model*

Looking at the generated topics for the new research papers depicted in Figure 6-19, we can clearly conclude that they do make sense and our NMF model is working quite well!

Visualizing Topic Models

We can also visualize our topic models in an interactive way in order to look at each topic and the theme conveyed by leveraging the pyLDAvis framework. Typically, dimension reduction techniques like MDS, PDA, and t-SNE are used to visualize the topics in a two-dimensional visual.

```
import pyLDAvis
import pyLDAvis.sklearn
import dill
import warnings

warnings.filterwarnings('ignore')
pyLDAvis.enable_notebook()

pyLDAvis.sklearn.prepare(nmf_model, cv_features, cv, mds='mmds')
```

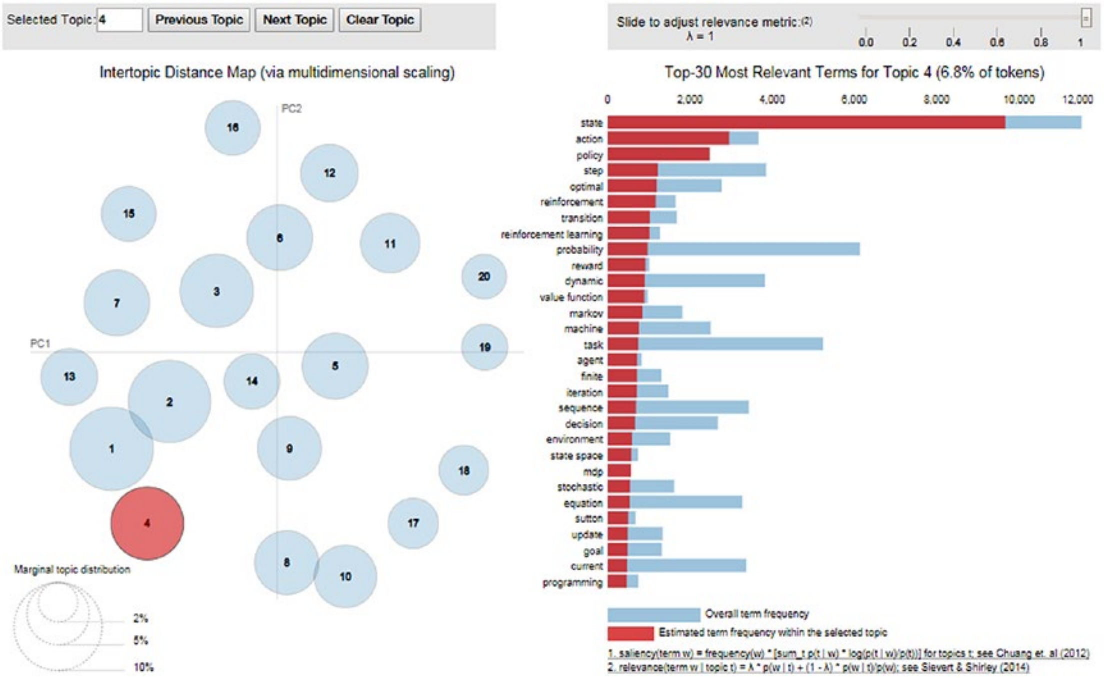

Selected Topic: 4
Previous Topic
Next Topic
Clear Topic
Slide to adjust relevance metric:(2)
λ = 1
0.0
0.2
0.4
0.6
0.8
1
Intertopic Distance Map (via multidimensional scaling)
PC2
PC1
16
12
15
6
11
20
7
3
19
13
14
5
2
1
9
18
4
17
8
10
Marginal topic distribution
2%
5%
10%
Top-30 Most Relevant Terms for Topic 4 (6.8% of tokens)
0
2,000
4,000
6,000
8,000
10,000
12,000
state
action
policy
step
optimal
reinforcement
transition
reinforcement learning
probability
reward
dynamic
value function
markov
machine
task
agent
finite
iteration
sequence
decision
environment
state space
mdp
stochastic
equation
sutton
update
goal
current
programming
Overall term frequency
Estimated term frequency within the selected topic
1. saliency(term w) = frequency(w) * [sum_t p(t | w) * log(p(t | w)/p(t))] for topics t; see Chuang et. al (2012)
2. relevance(term w | topic t) = λ * p(w | t) + (1 - λ) * p(w | t)/p(w); see Sievert & Shirley (2014)

Figure 6-20. *Visualizing topics from our NMF Model*

The visualization in Figure 6-20 is interactive in the Jupyter notebook and you can play around with it by checking out each topic, distribution of words, and topic distributions. We hope this gives you enough perspective of topic models to get started with modeling your own corpora.

Automated Document Summarization

We briefly talked about document summarization at the beginning of this chapter, when we mentioned extracting the gist from a large document or corpus so that it retains the core essence or meaning of the corpus. The idea of document summarization is a bit different from keyphrase extraction or topic modeling. In this case, the end result is still in the form of some document, but with a few sentences based on the length we might want the summary to be. This is similar to an abstract or an executive summary in a research paper. The main objective of automated document summarization is to perform this summarization without involving human input, except for running computer programs. Mathematical and statistical models help in building and automating the task of summarizing documents by observing their content and context.

There are two broad approaches to document summarization using automated techniques. They are described as follows:

- **Extraction-based techniques:** These methods use mathematical and statistical concepts like SVD to extract some key subset of the content from the original document such that this subset of content contains the core information and acts as the focal point of the entire document. This content can be words, phrases, or even sentences. The end result from this approach is a short executive summary of a couple of lines extracted from the original document. No new content is generated in this technique, hence the name *extraction-based.*
- **Abstraction-based techniques:** These methods are more complex and sophisticated. They leverage language semantics to create representations and use natural language generation (NLG) techniques where the machine uses knowledge bases and semantic representations to generate text on its own and create summaries just like a human would write them. Thanks to deep learning, we can implement these techniques easily but they require a lot of data and compute.

Much more research exists for extraction-based techniques since it is comparatively harder to build abstraction-based summarizers. Recently, substantial advances have been made in that area with regards to creating abstract summaries mimicking humans. Deep learning models, especially encoder-decoder architectures, have been very effective in summarizing text using abstractive methods. Implementing them is out of our current scope but we will be covering essentials of extractive text summarization with hands-on examples.

We use the description of a very popular role-playing game (RPG) *Skyrim* from Bethesda Softworks for summarization. The following is an excerpt from the document we will be summarizing (the full document is present in the Jupyter notebook for text summarization).

```
DOCUMENT = """
The Elder Scrolls V: Skyrim is an action role-playing video game developed
by Bethesda Game Studios and published by Bethesda Softworks. It is the
fifth main installment in The Elder Scrolls series, following The Elder
Scrolls IV: Oblivion. The game's main story revolves around the player
```

```
character's quest to defeat Alduin the World-Eater, a dragon who is
prophesied to destroy the world. The game is set 200 years after the events
of Oblivion and takes place in the fictional province of Skyrim. Over the
course of the game, the player completes quests and develops the character
by improving skills. The game continues the open-world tradition of
its predecessors by allowing the player to travel anywhere in the
game world at any time, and to ignore or postpone the main storyline
indefinitely. The team opted for a unique and more diverse open world than
Oblivion's Imperial Province of Cyrodiil, which game director and executive
producer Todd Howard considered less interesting by comparison. The game
was released to critical acclaim, with reviewers particularly mentioning
the character advancement and setting, and is considered to be one of the
greatest video games of all time.

The Elder Scrolls V: Skyrim is an action role-playing game, playable from
either a first or third-person perspective. The player may freely roam
over the land of Skyrim which is an open world environment consisting of
wilderness expanses, dungeons, cities, towns, fortresses, and villages.
...
...
A regeneration period limits the player's use of shouts in gameplay.

Skyrim is set around 200 years after the events of The Elder Scrolls IV:
Oblivion, although it is not a direct sequel. The game takes place in
Skyrim, a province of the Empire on the continent of Tamriel, amid a civil
war between two factions: the Stormcloaks, led by Ulfric Stormcloak, and
the Imperial Legion, led by General Tullius. The player character is a
Dragonborn, a mortal born with the soul and power of a dragon. Alduin, a
large black dragon who returns to the land after being lost in time, serves
as the game's primary antagonist. Alduin is the first dragon created by
Akatosh, one of the series' gods, and is prophesied to destroy and consume
the world.
"""
```

We need to do some basic preprocessing on this document to remove extra newlines. We can do this using the following code.

```
import re

DOCUMENT = re.sub(r'\n|\r', ' ', DOCUMENT)
DOCUMENT = re.sub(r' +', ' ', DOCUMENT)
DOCUMENT = DOCUMENT.strip()
```

Let's look at an implementation of document summarization by leveraging Gensim's summarization module. It is pretty straightforward, as depicted in the following code.

```
from gensim.summarization import summarize

print(summarize(DOCUMENT, ratio=0.2, split=False))
```

```
The game's main story revolves around the player character's quest to
defeat Alduin the World-Eater, a dragon who is prophesied to destroy the
world.
Over the course of the game, the player completes quests and develops the
character by improving skills.
The game continues the open-world tradition of its predecessors by allowing
the player to travel anywhere in the game world at any time, and to ignore
or postpone the main storyline indefinitely.
The player may freely roam over the land of Skyrim which is an open world
environment consisting of wilderness expanses, dungeons, cities, towns,
fortresses, and villages.
Each city and town in the game world has jobs that the player can engage
in, such as farming.
Over the course of the game, players improve their character's skills which
are numerical representations of their ability in certain areas.
Like other creatures, dragons are generated randomly in the world and will
engage in combat with NPCs, creatures and the player.
```

We can also limit the summarization based on word count instead of proportions by using the following code.

```
print(summarize(DOCUMENT, word_count=75, split=False))
```

```
The game's main story revolves around the player character's quest to defeat
Alduin the World-Eater, a dragon who is prophesied to destroy the world.
Over the course of the game, the player completes quests and develops the
character by improving skills.
The player may freely roam over the land of Skyrim which is an open world
environment consisting of wilderness expanses, dungeons, cities, towns,
fortresses, and villages.
```

I'm sure even if you have never played *Skyrim* before or read that huge block of text talking about the game, this summary gives you a pretty good idea what the game is all about. This is the power of text summarization, where using a few influential sentences, we can summarize the core theme of an entire document.

This summarization implementation from Gensim is based on a variation of a popular algorithm called *TextRank*. Now that we have seen how interesting text summarization can be, let's look at a couple of extraction-based summarization algorithms. We focus on the following two techniques:

- Latent Semantic Analysis
- TextRank

We first look at the concepts and math behind each technique, then implement them using Python, and finally test them on our toy document. Before we deep dive into the techniques, let's prepare our document by parsing and normalizing it.

Text Wrangling

We need to do some basic text wrangling or preprocessing on our document. Nothing too fancy, since the focus is on document summarization.

```
import nltk
import numpy as np
import re

stop_words = nltk.corpus.stopwords.words('english')


def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z\s]', '', doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = nltk.word_tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)

# get sentences in the document
sentences = nltk.sent_tokenize(DOCUMENT)

# normalize each sentence in the document
norm_sentences = normalize_corpus(sentences)
norm_sentences[:3]
```

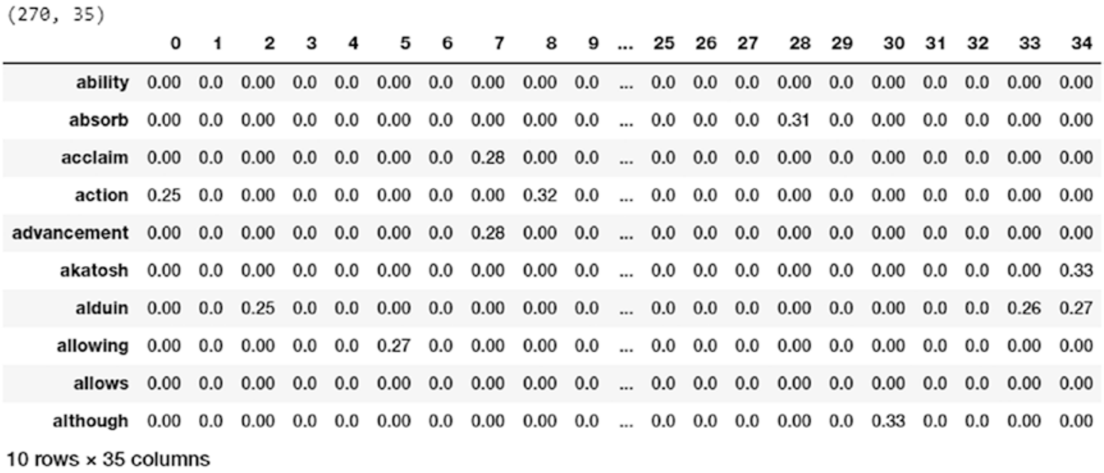
```
array(['elder scrolls v skyrim action roleplaying video game developed
bethesda game studios
         published bethesda softworks',
       'fifth main installment elder scrolls series following elder scrolls
       iv oblivion',
       'games main story revolves around player characters quest defeat
       alduin worldeater dragon
         prophesied destroy world'],
      dtype='<U183')
```

Our corpus is now preprocessed and normalized. Now we can leverage feature engineering to represent our text data in an efficient vectorized format.

Text Representation with Feature Engineering

We will be vectorizing our normalized sentences using the TF-IDF feature engineering scheme. We keep things simple and don't filter out any words based on document frequency. But feel free to try that out and maybe even leverage n-grams as features.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

tv = TfidfVectorizer(min_df=0., max_df=1., use_idf=True)
dt_matrix = tv.fit_transform(norm_sentences)
dt_matrix = dt_matrix.toarray()

vocab = tv.get_feature_names()
td_matrix = dt_matrix.T
print(td_matrix.shape)
pd.DataFrame(np.round(td_matrix, 2), index=vocab).head(10)
```



```
(270, 35)
```

	0	1	2	3	4	5	6	7	8	9	...	25	26	27	28	29	30	31	32	33	34
ability	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
absorb	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.31	0.0	0.00	0.0	0.0	0.00	0.00
acclaim	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.28	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
action	0.25	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.32	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
advancement	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.28	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
akatosh	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.33
alduin	0.00	0.0	0.25	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.26	0.27
allowing	0.00	0.0	0.00	0.0	0.0	0.27	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
allows	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.00	0.0	0.0	0.00	0.00
although	0.00	0.0	0.00	0.0	0.0	0.00	0.0	0.00	0.00	0.0	...	0.0	0.0	0.0	0.00	0.0	0.33	0.0	0.0	0.00	0.00

10 rows × 35 columns

Figure 6-21. *Visualizing topics from our NMF model*

The output in Figure 6-21 is our standard term-document matrix showing the TF-IDF weights of each term across the various documents. In our case, each document is a sentence from our original game description and is represented as a column. We can now start implementing document summarization with the two techniques mentioned earlier.

Latent Semantic Analysis

Here, we summarize our game description by utilizing document sentences. The terms in each sentence of the document have been extracted to form the term-document matrix, which we observed in Figure 6-21. We apply low-rank Singular Value

Decomposition to this matrix. The core principle behind Latent Semantic Analysis (LSA) is that in any document, there exists a latent structure among terms that are related contextually and hence should also be correlated in the same singular space.

The approach we follow in our implementation is taken from the popular paper published in 2004 by J. Steinberger and K. Jezek entitled, "Using Latent Semantic Analysis in Text Summarization and Summary Evaluation," which proposes some improvements over the excellent work done by Y. Gong, X. Liu, "Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis," which was published in 2001. We recommend you read these two papers if you are interested in gaining more in-depth knowledge about this technique.

The main idea in our implementation is to use SVD (recall $M = USV^T$) so that **U** and **V** are the orthogonal matrices and **S** is the diagonal matrix, which can also be represented as a vector of the singular values. The original matrix can be represented as a term-document matrix where the rows are terms and each column is a document, i.e., a sentence from our document in this case. The values can be any type of weighting like Bag of Words model-based frequencies, TF-IDFs, or binary occurrences.

We use our `low_rank_svd()` function to create a low rank matrix approximation for **M** based on the number of concepts, **k**, which will be our number of singular values. The same **k** columns from matrix **U** will point to the term vectors for each of the **k** concepts and in case of matrix **V**, the **k** rows based on the top **k** singular values point to sentence vectors. Once we have ***U***, ***S***, and V^T from the SVD for the top **k** singular values based on the number of concepts **k**, we perform the following computations. Remember that the input parameters we need are the number of concepts **k** and the number of sentences **n** that we want the final summary to contain.

1. Get the sentence vectors from the matrix **V** (**k** rows).
2. Get the top **k** singular values from **S**.
3. Apply a threshold-based approach to remove singular values that are less than half of the largest singular value if any exist. This is a heuristic and you can play around with this value if you want. Mathematically, it's $S_i = 0 \; iff \; S_i < \frac{1}{2} S$.
4. Multiply each term sentence column from **V** squared with its corresponding singular value from **S**, also squared, to get sentence weights per topic.

5. Compute the sum of the sentence weights across the topics and take the square root of the final score to get the salience scores for each sentence in the document.

The salience score computations for each sentence can be mathematically represented as follows:

$$SS = \sqrt{\sum_{i=1}^{k} S_i V_i^T}$$

where *SS* denotes the saliency score for each sentence by taking the dot product between the singular values and the sentence vectors from V^T. Once we have these scores, we sort them in descending order, pick the top **n** sentences corresponding to the highest scores, and combine them to form our final summary based on the order in which they were present in the original document.

Let's implement our algorithm in Python. The first step is to select the number of sentences, **n**, that our summary will contain. Given we have around 35 sentences, we set our summary to contain eight sentences. The total number of topics or concepts, k, is set to three considering we have extracted the game description from the summary, gameplay, and plot sections from the original review in Wikipedia. Then we perform low-rank SVD.

```
num_sentences = 8
num_topics = 3

u, s, vt = low_rank_svd(td_matrix, singular_count=num_topics)
print(u.shape, s.shape, vt.shape)
term_topic_mat, singular_values, topic_document_mat = u, s, vt
```

```
(270, 3) (3,) (3, 35)
```

Next, we apply a threshold to set any of the existing singular values to 0, which is less than half of the largest singular value.

```
# remove singular values below threshold
sv_threshold = 0.5
min_sigma_value = max(singular_values) * sv_threshold
singular_values[singular_values < min_sigma_value] = 0
```

We can now leverage the formula we described earlier to compute the sentence saliency scores for each sentence (document) in our game description.

```
salience_scores = np.sqrt(np.dot(np.square(singular_values),
                                 np.square(topic_document_mat)))
salience_scores
```

```
array([0.53291263, 0.61639562, 0.60427539, 0.52307109, 0.50141128,
       0.32352969, 0.1506046 , 0.25383436, 0.60567083, 0.35902104,
       0.22562997, 0.34608934, 0.15781555, 0.40522541, 0.24505982,
       0.19874104, 0.39317895, 0.45392878, 0.31638528, 0.47353378,
       0.18348908, 0.45731421, 0.13929749, 0.38932101, 0.36829067,
       0.57822992, 0.40853736, 0.26260062, 0.38904585, 0.32776714,
       0.67662776, 0.21866561, 0.34687796, 0.3234621 , 0.46107093])
```

Now it is just a matter of selecting the top sentences based on their saliency score and displaying the summary of our game description.

```
top_sentence_indices = (-salience_scores).argsort()[:num_sentences]
top_sentence_indices.sort()
print('\n'.join(np.array(sentences)[top_sentence_indices]))
```

```
The Elder Scrolls V: Skyrim is an action role-playing video game developed
by Bethesda Game Studios and published by Bethesda Softworks.
It is the fifth main installment in The Elder Scrolls series, following The
Elder Scrolls IV: Oblivion.
The game's main story revolves around the player character's quest to defeat
Alduin the World-Eater, a dragon who is prophesied to destroy the world.
The game is set 200 years after the events of Oblivion and takes place in
the fictional province of Skyrim.
Over the course of the game, the player completes quests and develops the
character by improving skills.
The Elder Scrolls V: Skyrim is an action role-playing game, playable from
either a first or third-person perspective.
Skyrim is the first entry in The Elder Scrolls to include dragons in the
game's wilderness.
Skyrim is set around 200 years after the events of The Elder Scrolls IV:
Oblivion, although it is not a direct sequel.
```

Thus, you can see how a few matrix transformations give us a concise and excellent summarized document that covers the main aspects of our game description. This concludes our discussion of Latent Semantic Analysis. We move on to the next technique for extraction-based document summarization.

TextRank

The TextRank summarization algorithm internally uses the popular PageRank algorithm, which is used by Google for ranking websites and pages. This is used by the Google search engine when providing relevant web pages based on search queries. To understand TextRank better, we need to understand some of the concepts surrounding PageRank. The core algorithm in PageRank is a graph-based scoring or ranking algorithm, where pages are scored or ranked based on their importance. Websites and pages contain further links embedded in them which link to more pages having more links and this continues across the Internet. This can be represented as a graph-based model where vertices indicate the web pages and edges indicate links among them. This can be used to form a voting or recommendation system such so when one vertex links to another one in the graph it is basically casting a vote. Vertex importance is decided not only on the number of votes or edges but also the importance of the vertices that are connected to it and their importance. This helps determine the score or rank of each vertex or page. This is evident in Figure 6-22.

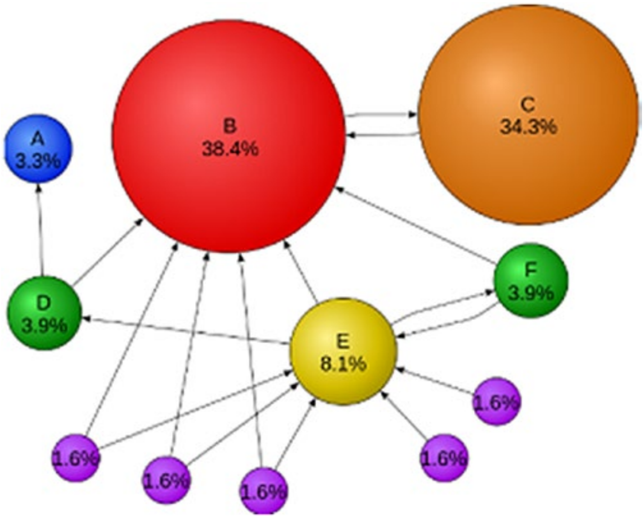


Figure 6-22. *PageRank scores for a simple network*

From Figure 6-22, we can see that vertex denoting Page C has a higher score than Page E even if it has fewer edges compared to Page E, because Page B is an important page connected to Page C. Thus, we can formally define PageRank as follows. Consider a directed graph represented as $G = (V, E)$ such that **V** represents the set of vertices or pages and **E** represents the set of edges or links. **E** is a subset of $V \times V$. Assuming we have a given page V_i for which we want to compute the PageRank, we can mathematically define it as follows:

$$PR(V_i) = (1-d) + d \times \sum_{j \in In(V_i)} \frac{PR(V_j)}{|Out(V_j|}$$

where for the vertex/page V_i we have $PR(V_i)$, which indicates the PageRank score. $In(V_i)$ represents the set of pages that point to this vertex/page, $Out(V_i)$ represents the set of pages that the vertex/page V_i points to, and d is the damping factor and usually has a value between 0 to 1 (ideally, it is set to 0.85).

Coming back to the TextRank algorithm, when summarizing a document, we will have sentences, keywords, or phrases as the vertices of the algorithm based on the type of summarization we are trying to do. We might have multiple links between these vertices. The modification that we make from the original PageRank algorithm is to have a weight coefficient (say w_{ij}) between the edge connecting two vertices V_i and V_j such that this weight indicates the strength of this connection between them. Thus we now formally define the new function for computing TextRank of vertices as follows:

$$TR(V_i) = (1-d) + d \times \sum_{V_j \in In(V_i)} \frac{w_{ji}\, TR(V_j)}{\sum_{V_k \in Out(V_j)} w_{jk}}$$

where *TR* indicates the weighted PageRank score for a vertex now defined as the TextRank for that vertex. Thus, we can formulate the algorithm and depict the main steps we will follow. They are defined as follows:

1. Tokenize and extract sentences from the document to be summarized.
2. Decide on the number of sentences, **k**, that we want in the final summary

3. Build a document-term feature matrix using weights like TF-IDF or Bag of Words.
4. Compute a document similarity matrix by multiplying the matrix by its transpose.
5. Use these documents (sentences in our case) as the vertices and the similarities between each pair of documents as the weight or score coefficient we talked about earlier and feed them to the PageRank algorithm.
6. Get the score for each sentence.
7. Rank the sentences based on score and return the top **k** sentences.

Since we already have our document-term feature matrix defined when we performed document summarization using LSA, we reuse that matrix, which is stored in the `dt_matrix` variable. The next step is to compute the document similarity matrix.

```
similarity_matrix = np.matmul(dt_matrix, dt_matrix.T)
print(similarity_matrix.shape)
np.round(similarity_matrix, 3)
```

```
(35, 35)
array([[1.   , 0.182, 0.   , ..., 0.   , 0.   , 0.   ],
       [0.182, 1.   , 0.05 , ..., 0.   , 0.   , 0.084],
       [0.   , 0.05 , 1.   , ..., 0.101, 0.165, 0.319],
       ...,
       [0.   , 0.   , 0.101, ..., 1.   , 0.066, 0.069],
       [0.   , 0.   , 0.165, ..., 0.066, 1.   , 0.123],
       [0.   , 0.084, 0.319, ..., 0.069, 0.123, 1.   ]])
```

Now we construct the connected graph among all the sentences from our toy document by using the document similarity scores and the documents themselves as the vertices. We use the `networkx` library to help us plot this graph. Remember that each document is a sentence in our case and will also be the vertices in the graph.

```
import networkx
```

```
# build the similarity graph
similarity_graph = networkx.from_numpy_array(similarity_matrix)
similarity_graph

<networkx.classes.graph.Graph at 0x1baf8a352b0>

# view the similarity graph
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(12, 6))
networkx.draw_networkx(similarity_graph, node_color='lime')
```

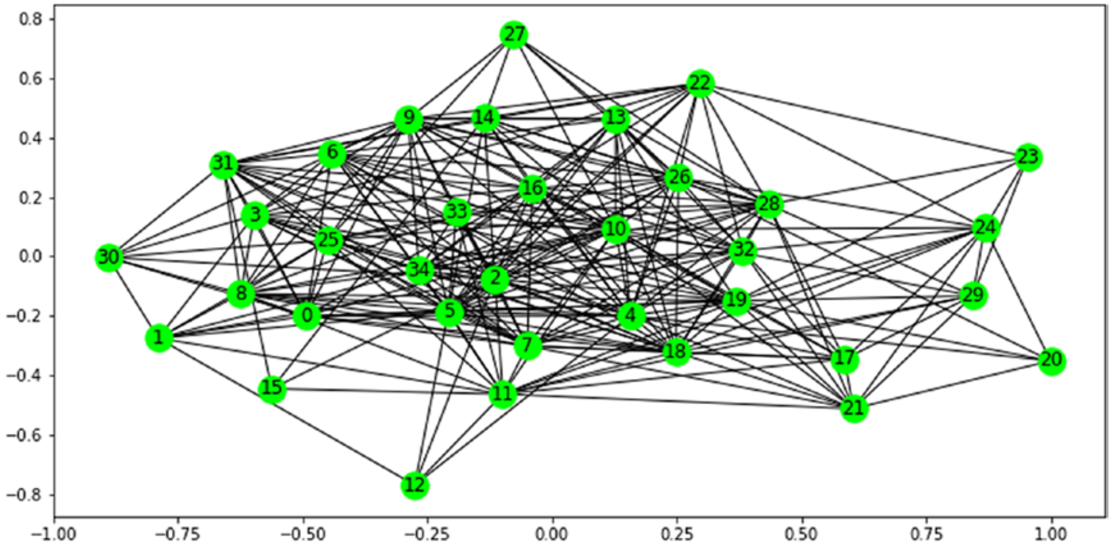


Figure 6-23. *Similarity graph showing connections between sentences*

From Figure 6-23, we can see how the sentences of our toy document are now linked to each other based on document similarities. The graph shows how well connected some sentences are to others. We now compute the PageRank scores for all the sentences and build our summary using the top eight sentences.

```
# compute pagerank scores for all the sentences
scores = networkx.pagerank(similarity_graph)
ranked_sentences = sorted(((score, index) for index, score
                                              in scores.items()),
                          reverse=True)
ranked_sentences[:10]
```

```
[(0.03729704979721473, 2),
 (0.03490843547537587, 25),
 (0.03460086870923609, 4),
 (0.03240744530656926, 8),
 (0.03218748996523769, 28),
 (0.03183673426880143, 11),
 (0.031566658693076226, 26),
 (0.03150616293402057, 3),
 (0.031376143577383796, 5),
 (0.031123481531894214, 16)]
```

Once each sentence has been ranked (based on the sentence indices you can see in the preceding output), we sort them based on their score. We can then easily identify the top eight sentences to form our summary.

```
# get the top sentence indices for our summary
top_sentence_indices = [ranked_sentences[index][1]
                        for index in range(num_sentences)]
top_sentence_indices.sort()

# construct the document summary
print('\n'.join(np.array(sentences)[top_sentence_indices]))
```

```
The game's main story revolves around the player character's quest to defeat
Alduin the World-Eater, a dragon who is prophesied to destroy the world.
The game is set 200 years after the events of Oblivion and takes place in
the fictional province of Skyrim.
Over the course of the game, the player completes quests and develops the
character by improving skills.
The Elder Scrolls V: Skyrim is an action role-playing game, playable from
either a first or third-person perspective.
The game's main quest can be completed or ignored at the player's
preference after the first stage of the quest is finished.
```

```
Skyrim is the first entry in The Elder Scrolls to include dragons in the
game's wilderness.
Like other creatures, dragons are generated randomly in the world and will
engage in combat with NPCs, creatures and the player.
The player character can absorb the souls of dragons in order to use
powerful spells called "dragon shouts" or "Thu'um".
```

We finally get our desired summary by using the TextRank algorithm. The content is also quite meaningful and you will see a lot of similarity with the Gensim output, which is based on a variation of the TextRank algorithm.

You can see from this output that we were successfully able to summarize our product description. This short summary depicts the core essence of the product description like the name of the game and its various features regarding its gameplay and plot. This concludes our discussion of automated text summarization. We encourage you to try these techniques on more documents and test it with various parameters. Consider parameters like more topics and different features, and maybe even explore deep learning based techniques for abstractive text summarization.

Summary

In this chapter, we covered some interesting areas in natural language processing and text analytics with regard to information extraction, document summarization, and topic modeling. We started with an overview of the evolution of information being generated in the world and learned about concepts like information overload leading to the need for text summarization and information retrieval. We talked about the various ways we can extract key information from textual data and ways of summarizing large documents. We also covered important mathematical concepts like Singular Value Decomposition and low rank matrix approximation and utilized them in several of our algorithms.

We covered three approaches to reducing information overload, which included keyphrase extraction, topic models, and automated document summarization. Keyphrase extraction included methods like collocations and weighted tagged term based approaches for getting keyphrases or terms from corpora.

We built several topic modeling techniques including Latent Semantic Indexing, latent dirichlet allocation and the very recently implemented non-negative matrix factorization using Gensim and Scikit-Learn. We also tuned our topic models and showcased a method to find the optimal number of topics. Besides that, we looked at effective ways of interpreting and understanding topic modeling results. Finally, we looked at two extraction-based techniques for automated document summarization—Latent Semantic Analysis and TextRank. We implemented each method and observed results on real-world data to get a good idea of how these methods worked and how effective simple mathematical operations can be in generating actionable insights.

CHAPTER 7

Text Similarity and Clustering

In the previous chapters, we covered several techniques to analyze text and extract interesting insights. We looked at supervised machine learning techniques, which are used to categorize text documents into several assumed categories. Unsupervised techniques like topic models and document summarization were also covered, which involved trying to retrieve key themes and information from large text documents and corpora.

In this chapter, we look at several interesting techniques and use cases that leverage unsupervised learning and information retrieval concepts. If you refresh your memory about Chapter 5, text categorization is indeed an interesting problem that has several applications, most notably in news articles categorization and e-mail classification. But one constraint in text classification is that we need a good amount of training data with manually labeled categories, since we use supervised learning algorithms to build our classification model. Building a labeled dataset is definitely not easy because you need a sizeable amount of training data. For this, we need to spend time and manual effort in labeling data, building the model, and then using it to classify new documents. Usually any enterprise might not have enough time to invest in this (even though the benefits can be ten-fold!). Can we instead make the machine do this task? Maybe to an extent! This chapter specifically looks at the content of text documents, analyzing their similarity using various measures, and clustering similar documents together.

Text data is unstructured and highly noisy. We get the benefits of well labeled training data and supervised learning when performing text classification. However, document clustering is an unsupervised learning process, whereby we are trying to segment and categorize documents into separate categories by making the machine learn about the various text documents, their features, similarities, and differences. This makes document clustering more challenging, albeit interesting.

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_7

Consider having a corpus of documents that talks about various different concepts and ideas. Humans are wired in such a way that we use our learning from the past and apply it to distinguish concepts. For example, the sentence "The fox is smarter than the dog" is more similar to "The dog is faster than the fox" as compared to "Python is an excellent programming language". We can easily spot and intuitively determine specific key phrases like Python, fox, dog, programming, and so on, which helps us determine which sentences or documents are more similar. But can we do this programmatically?

In this chapter, we focus on several concepts related to text similarity—distance metrics and unsupervised machine learning algorithms—to answer the following questions.

- How do we measure similarity between terms and documents?
- How can we use distance measures to find the most relevant documents?
- When is a distance measure a metric?
- How can we build a recommender system from text similarity?
- How do we group similar documents?

While we focus on trying to answer these questions, we also cover essential concepts and information needed to understand various techniques for solving these problems. We also use some practical examples to illustrate concepts related to text similarity, distance metrics, and document clustering. We depict these using some interesting case studies of building a movie recommender using document similarity and cluster similar movies together!

Many of these techniques can be combined with some of the techniques we learned previously and vice versa. For example, concepts of text representation with feature engineering and text similarity using distance metrics and features are also used to build document clusters. You can also use features from topic models to measure text similarity.

Besides this, clustering can give us a feel for the possible groups or categories that our data might consist of, based on similar patterns and attributes. This can then be plugged in to other systems like supervised classification systems. The possibilities are indeed endless! In this chapter, we first cover some important concepts related to distance measures, metrics, and unsupervised learning. Once the basics have been covered, our objective is to understand and analyze term similarity, document similarity, recommendations, and finally document clustering. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at `https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition`.

Essential Concepts

Our main objective in this chapter is to understand text similarity and clustering. Before moving on to the actual techniques and algorithms, this section discusses some important concepts related to information retrieval, document similarity measures, and machine learning. Even though some of these concepts might be familiar to you from the previous chapters, a brief refresher will be useful. Without further ado, let's get started.

Information Retrieval (IR)

Information retrieval is defined as the process of retrieving relevant sources of information from a corpus or set of entities that hold information based on some demand. A demand can in the form of a query or search that users enter in a search engine. They then get relevant search items pertaining to their query. In fact, search engines are the most popular application of IR. The relevancy of documents with information compared to the demand can be measured in several ways. This includes looking for specific keywords from the search text or using similarity measures to see the similarity rank or score of the documents with respect to the entered query. This makes is quite different from string matching or matching regular expressions because more than often the words in a search string can have different order, context, and semantics in the collection of documents (entities), and these words can even have multiple different resolutions or possibilities based on synonyms, antonyms, and negation modifiers.

Feature Engineering

Feature engineering or feature extraction is something that you know about quite well by now. We covered this in detail in Chapter 4. Methods like Bag of Words, TF-IDF, and word embedding models are typically used to represent documents in the form of numeric vectors so that applying mathematical or machine learning techniques becomes much easier. You can use various document representations using these feature extraction techniques or even map each letter or a word to a corresponding unique numeric identifier.

Similarity Measures

Similarity measures are used in text similarity analysis and clustering. Any similarity or distance measure measures the degree of closeness between two entities, which can be any text format like documents, sentences, or even terms. This measure of similarity can be useful in identifying similar entities and distinguishing clearly different entities from each other. Similarity measures are very effective and sometimes choosing the right measure can make a lot of difference in the performance of your final analytics system. Various scoring or ranking algorithms have also been invented based on these distance measures. There are two main factors that determine the degree of similarity between entities. They are as follows:

- Inherent properties or features of the entities
- Measure formula and properties

There are several distance measures that measure similarity and we will be covering several of them in future sections. However, an important point to remember is that all distance measures of similarity are not distance metrics of similarity. The excellent paper by A. Huang entitled "Similarity Measures for Text Document Clustering" talks about this in detail. Consider a distance measure d and two entities (let us consider them to be documents in our context), x and y. The distance between x and y is used to determine the degree of similarity between them. It can be represented as $d(x, y)$ but the measure d is called a *distance metric of similarity* if and only if satisfies the following four conditions.

- The distance measured between any two entities, say **x** and **y**, must be always non-negative, i.e., $d(x, y) \geq 0$.
- The distance between two entities should always be zero if and only if they are identical, i.e., $d(x, y) \geq 0 \ iff \ x = y$.

- This distance measure should always be symmetric, which means that the distance from **x** to **y** is always the same as the distance from **y** to **x**. Mathematically, this is represented as $d(x, y) = d(y, x)$.
- This distance measure should satisfy the triangle inequality property, which can be mathematically represented $d(x, z) \leq d(x, y) + d(y, z)$.

This tells us important criteria and gives us a good framework that we can use to check if a distance measure can be used as a distance metric for measuring similarity. Going into more details would be currently out of the scope, but you might be interested in knowing that the very popular KL-divergence measure also known as Kullback-Leibler divergence is a distance measure that violates the third property, where this measure is asymmetric. Hence, it does not make sense to use this as a measure of similarity for text documents. Otherwise, it's extremely useful in differentiating between various distributions and patterns.

Unsupervised Machine Learning Algorithms

These refer to the family of machine learning algorithms that try to discover latent hidden structures and patterns in data from their various attributes and features. Besides this, several unsupervised learning algorithms are also used to reduce the feature space, which is often of a higher dimension to one with a lower dimension. The data on which these algorithms operate is essentially unlabeled data, so it does not have any predetermined category or class. We apply these algorithms with the intent of find patterns and distinguishing features that might help us in grouping various data points into groups or clusters. These algorithms are popularly known as *clustering algorithms.* Even topic models covered in the previous chapter belong to the unsupervised learning family of algorithms. This concludes our discussion on the important concepts and background information necessary for this chapter. We now move on to text normalization and feature extraction, where we introduce a few concepts specific to this chapter.

Text Similarity

The main objective of text similarity is to analyze and measure how close two entities of text are to each other. These entities of text can be simple tokens or terms like words or whole documents, which may include sentences or paragraphs of text. There are various ways to analyze text similarity and we can classify the intent of text similarity broadly into the following two areas.

- **Lexical similarity:** This involves observing the contents of the text documents with regards to its syntax, structure, and content and measuring their similarity based on these parameters.
- **Semantic similarity:** This involves determining the semantics, meaning, and context of the documents and then determining how close they are to each other. Dependency grammars and entity recognition are handy tools that can help in this. We covered word embedding methods in detail in Chapter 4, which help in capturing semantic information.

We cover lexical similarity in this chapter. Distance metrics are typically used to measure similarity scores between text entities and we mainly cover the following two broad areas of text similarity:

- **Term similarity:** Similarity between individual tokens or words
- **Document similarity:** Similarity between entire text documents

The idea is to implement several distance metrics and see how we can measure and analyze similarity among simple words. Then we look at how things change when we measure similarity among groups of individual words.

Analyzing Term Similarity

We will start by analyzing term similarity, or similarity between individual word tokens, to be more precise. Even though this is not used a lot in practical applications, this can be an excellent starting point to understanding text similarity. Of course, several applications and use cases like autocompleters, spell check, and correctors use these techniques to correct misspelled terms. We saw a fair bit of that during our spell check implementation in Chapter 3! Here we take a few words and measure the similarity between then using different word representations as well as distance metrics. The word representations we use are as follows:

- Character vectorization
- Bag of characters vectorization

For character vectorization, it is an extremely simple process of just mapping each character of the term to a corresponding unique number. We can do that using the function depicted in the following snippet.

```
import numpy as np

def vectorize_terms(terms):
    terms = [term.lower() for term in terms]
    terms = [np.array(list(term)) for term in terms]
    terms = [np.array([ord(char) for char in term])
                for term in terms]
    return terms
```

This function takes a list of words or terms and returns the corresponding character vectors for the words. To demonstrate this, we use a total of four example terms and compute the similarity among them shortly.

```
root = 'Believe'
term1 = 'beleive'
term2 = 'bargain'
term3 = 'Elephant'

terms = [root, term1, term2, term3]
terms

['Believe', 'beleive', 'bargain', 'Elephant']
```

Let's now perform character vectorization on each of these strings (list of character tokens) and view their representation in the form of a data frame.

```
# Character vectorization
term_vectors = vectorize_terms(terms)

# show vector representations
vec_df = pd.DataFrame(term_vectors, index=terms)
print(vec_df)
```

```
            0    1    2    3    4    5    6      7
Believe    98  101  108  105  101  118  101    NaN
beleive    98  101  108  101  105  118  101    NaN
bargain    98   97  114  103   97  105  110    NaN
Elephant  101  108  101  112  104   97  110  116.0
```

Thus you can see how we can easily transform each text term into a corresponding numeric vector representation. Note that the NaN values indicate that those strings are one character shorter, as compared to the last string, which is one character longer. We now use several distance metrics to compute similarity between the root word and the other three words, as mentioned in the previous snippet. There are a lot of distance metrics out there that you can use to compute and measure similarities. We cover the following five metrics in this section.

- Hamming distance
- Manhattan distance
- Euclidean distance
- Levenshtein Edit distance
- Cosine distance and similarity

We look at the concepts for each distance metric and use the power of NumPy arrays to implement the necessary computations and mathematical formulae. Once we do that, we put them in action by measuring the similarity of our example terms. Before we do this, we set up some necessary variables by storing the root term, the other terms with which its similarity will be measured, and their various vector representations using the following snippet.

```
root_term = root
other_terms = [term1, term2, term3]

root_term_vec = vec_df[vec_df.index == root_term].dropna(axis=1).values[0]
other_term_vecs = [vec_df[vec_df.index == term].dropna(axis=1).values[0]
                      for term in other_terms]
```

We are now ready to start computing similarity metrics and will use these terms and their vector representations to measure similarities.

Hamming Distance

The *Hamming distance* is a very popular distance metric used frequently in information theory and communication systems. It is the distance measured between two strings under the assumption that they are of equal length. Formally it is defined as the number of positions that have different characters or symbols between two strings of equal length. Considering two terms **u** and **v** of length **n**, we can mathematically denote Hamming distance as follows:

$$hd(u,v) = \sum_{i=1}^{n} (u_i \neq v_i)$$

You can also normalize it if you want by dividing the number of mismatches by the total length of the terms. This gives the normalized hamming distance, which is represented as follows:

$$norm\_hd(u,v) = \frac{\sum_{i=1}^{n} (u_i \neq v_i)}{n}$$

Note that **n** denotes the length of the terms. The following function computes the Hamming distance between two terms and has the capability to compute the normalized distance.

```
def hamming_distance(u, v, norm=False):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    return (u != v).sum() if not norm else (u != v).mean()
```

We can measure the Hamming distance between our root term and the other terms using the following code snippet.

```
# compute Hamming distance
for term, term_vector in zip(other_terms, other_term_vecs):
    print('Hamming distance between root: {} and term: {} is {}'.
    format(root_term, term, hamming_distance(root_term_vec,term_vector,
    norm=False)))
```

```
Hamming distance between root: Believe and term: beleive is 2
Hamming distance between root: Believe and term: bargain is 6
```

```
Traceback (most recent call last):
  File "<ipython-input-115-3391bd2c4b7e>", line 4, in <module>
    hamming_distance(root_vector, vector_term, norm=False))
ValueError: The vectors must have equal lengths.
```

```
# compute normalized Hamming distance
for term, term_vector in zip(other_terms, other_term_vecs):
    print('Normalized Hamming distance between root: {} and term: {}
    is {}'.format(root_term, term, round(hamming_distance(root_term_vec,
    term_vector, norm=True), 2)))
```

```
Normalized Hamming distance between root: Believe and term: beleive is 0.29
Normalized Hamming distance between root: Believe and term: bargain is 0.86
Traceback (most recent call last):
  File "<ipython-input-117-7dfc67d08c3f>", line 4, in <module>
    round(hamming_distance(root_vector, vector_term, norm=True), 2))
ValueError: The vectors must have equal lengths
```

You can see from the output that the terms “Believe” and “beleive” are most similar, with a Hamming distance of 2 or 0.29, compared to the term “bargain,” giving scores of 6 or 0.86 (the smaller the score more, the similar the terms). Likewise, the term “Elephant” throws an exception because its length is different than the root term (“Believe”). The Hamming distance can't be computed since the strings aren’t equal length.

Manhattan Distance

The *Manhattan distance* metric is similar to the Hamming distance conceptually where, instead of counting the number of mismatches, we subtract the difference between each pair of characters at each position of the two strings. Formally, the Manhattan distance is also known as city block distance, L1 norm, or taxicab metric, and it is defined as the distance between two points in a grid based on strictly horizontal or vertical paths. This is instead of the diagonal distance conventionally calculated by the Euclidean distance metric. Mathematically it can be denoted as follows:

$$md(u,v) = \|u - v\|_1 = \sum_{i=1}^{n} |u_i - v_i|$$

where **u** and **v** are the two terms of length **n**. The same assumption of the two terms having equal length from the Hamming distance holds good here. We can also compute the normalized Manhattan distance by dividing the sum of the absolute differences by the term length. This can be denoted by

$$norm\_md(u,v) = \frac{\|u - v\|_1}{n} = \frac{\sum_{i=1}^{n} |u_i - v_i|}{n}$$

where **n** is the length of each of the terms **u** and **v**. The following function helps us implement the Manhattan distance with the capability to also compute the normalized Manhattan distance.

```
def manhattan_distance(u, v, norm=False):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    return abs(u - v).sum() if not norm else abs(u - v).mean()
```

We will now compute the Manhattan distance between our root term and the other terms using the function depicted in the following code snippet.

```
# compute Manhattan distance
for term, term_vector in zip(other_terms, other_term_vecs):
    print('Manhattan distance between root: {} and term: {} is {}'.
    format(root_term, term,manhattan_distance(root_term_vec, term_vector,
    norm=False)))
```

```
Manhattan distance between root: Believe and term: beleive is 8
Manhattan distance between root: Believe and term: bargain is 38
Traceback (most recent call last):
  File "<ipython-input-120-b228f24ad6a2>", line 4, in <module>
    manhattan_distance(root_vector, vector_term, norm=False))
ValueError: The vectors must have equal lengths.
```

```
# compute normalized Manhattan distance
for term, term_vector in zip(other_terms, other_term_vecs):
    print('Normalized Manhattan distance between root: {} and term: {} is
    {}'.format(root_term, term, round(manhattan_distance(root_term_vec,
    term_vector, norm=True), 2)))
```

```
Normalized Manhattan distance between root: Believe and term: beleive is 1.14
Normalized Manhattan distance between root: Believe and term: bargain is 5.43
Traceback (most recent call last):
  File "<ipython-input-122-d13a48d56a22>", line 4, in <module>
    round(manhattan_distance(root_vector, vector_term, norm=True),2))
ValueError: The vectors must have equal lengths.
```

From these results, you can see that as expected, "Believe" and "beleive" are most similar, with a score of 8 or 1.14, as compared to "bargain," which gives a score of 38 or 5.43. The term "Elephant" yields an error because it has a different length compared to the base term.

Euclidean Distance

We briefly mentioned the *Euclidean distance* when comparing it to the Manhattan distance in the earlier section. Formally, the Euclidean distance is also known as the Euclidean norm, L2 norm, or L2 distance. It's defined as the shortest straight-line distance between two points. Mathematically, this can be denoted as follows:

$$ed(u,v) = \| u - v \|_2 = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2}$$

where the two points **u** and **v** are vectorized text terms in our scenario with a length of **n**. The following function helps us compute the Euclidean distance between two terms.

```
def euclidean_distance(u, v):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    distance = np.sqrt(np.sum(np.square(u - v)))
    return distance
```

We can now compare the Euclidean distance among our terms by using this function, as depicted in the following code snippet.

```
# compute Euclidean distance
for term, term_vector in zip(other_terms, other_term_vecs):
    print('Euclidean distance between root: {} and term: {} is {}'.format(root_
    term, term, round(euclidean_distance(root_term_vec, term_vector), 2)))
```

```
Euclidean distance between root: Believe and term: beleive is 5.66
Euclidean distance between root: Believe and term: bargain is 17.94
Traceback (most recent call last):
  File "<ipython-input-132-90a4dbe8ce60>", line 4, in <module>
    round(euclidean_distance(root_vector, vector_term),2))
ValueError: The vectors must have equal lengths.
```

From these outputs, you can see that the terms “Believe” and “beleive” are the most similar, with a score of 5.66, compared to “bargain” with a score of 17.94. Again, the “Elephant” string throws a `ValueError` because it is a different length. So far, all the distance metrics we used work on strings of the same length and fail when they are not of equal length. So how do we deal with this problem? We now look at a couple of distance metrics that measure similarity even with strings of unequal length.

Levenshtein Edit Distance

The *Levenshtein Edit distance,* often known as just the *Levenshtein distance,* belongs to the family of edit distance based metrics and is used to measure the distance between two sequence of strings based on their differences, similar to the concept behind the Hamming distance. The Levenshtein Edit distance between two terms can be defined as the minimum number of edits needed in the form of additions, deletions, or substitutions to change or convert one term to the other. These substitutions are character-based substitutions, where a single character can be edited in a single operation. As mentioned, the length of the two terms need not be equal. Mathematically, we can represent the Levenshtein Edit distance between two terms as $ld_{u,\,v}(|u|, |v|)$, where **u** and **v** are our two terms and $|u|$ and $|v|$ are their lengths. This distance can be represented by the following formula:

$$ld_{u,v}(i,j) = \left\{ \begin{array}{ll} \max(i,j) & \mathit{if} \min(i,j) = 0 \\ \min \left\{ \begin{array}{c} ld_{u,v}(i-1,j)+1 \\ ld_{u,v}(i,j-1)+1 \\ ld_{u,v}(i-1,j-1)+C_{u_i \neq v_j} \end{array} \right\} & \mathit{otherwise} \end{array} \right\}$$

where **i** and **j** are basically indices for the terms **u** and **v**. The third equation in the minimum includes a cost function denoted by $C_{u_i \neq v_j}$ and it has the following conditions:

$$C_{u_i \neq v_j} = \begin{Bmatrix} 1 & \mathit{if}\ u_i \neq v_j \\ 0 & \mathit{if}\ u_i = v_j \end{Bmatrix}$$

This denotes the indicator function, which depicts the cost associated with two characters being matched for the two terms (the equation represents the match or mismatch operation). The first equation in the minimum determines the deletion operation and the second equation determines the insertion operation.

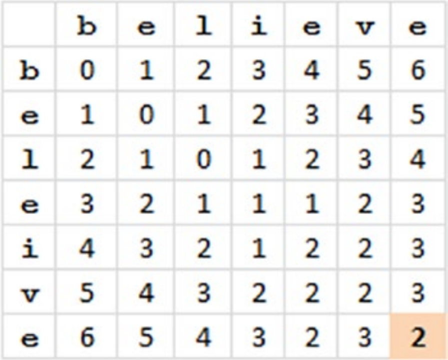
The function $ld_{u,\,v}(i,j)$ thus covers all the three operations of insertion, deletion, and addition. It also denotes the Levenshtein distance, as measured between the first **i** characters for the term u and the first **j** characters of the term **v**. There are also several interesting boundary conditions with regards to the Levenshtein Edit distance. They are as follows:

- The minimum value that the edit distance can take between two terms is the difference in length of the two terms
- The maximum value of the edit distance between two terms can be the length of the term that's larger
- If the two terms are equal, the edit distance is zero
- The Hamming distance between two terms is an upper bound for the Levenshtein Edit distance if and only if the two terms have equal lengths
- This being a distance metric, it also satisfies the triangle inequality property, which we discussed earlier when we talked about distance metrics

There are various ways of implementing Levenshtein distance computations for terms. Here we start with an example of two of our terms. Considering the root term "believe" and another term, "beleive" (we ignore case in our computations), the edit distance would be 2 because we would need the following two operations.

- bel**e**ive → bel**i**ive (substitution of **e** to **i**)
- beli**i**ve → beli**e**ve (substitution of **i** to **e**)

To implement this, we build a matrix that computes the Levenshtein distance between all the characters of both terms by comparing each character of the first term with the characters of the second term. For computation, we follow a dynamic programming approach, in order to get the edit distance between the two terms based on the last computed value. For the two terms, the Levenshtein Edit distance matrix that our algorithm should generate is depicted in Figure 7-1.



	b	e	l	i	e	v	e
b	0	1	2	3	4	5	6
e	1	0	1	2	3	4	5
l	2	1	0	1	2	3	4
e	3	2	1	1	1	2	3
i	4	3	2	1	2	2	3
v	5	4	3	2	2	2	3
e	6	5	4	3	2	3	2

Figure 7-1. *Levenshtein Edit distance matrix between terms*

You can see from the Figure 7-1 that the edit distances are computed for each pair of characters in the terms and the final edit distance value (which is highlighted in the figure) gives us the actual edit distance between the two terms. This algorithm is also known as the *Wagner-Fischer algorithm* and is available in the paper by R. Wagner and M. Fischer entitled, "The String-to-String Correction Problem," which you can refer to if you are interested in the details. The pseudocode is depicted in the following snippet, courtesy of the paper.

```
function levenshtein_distance(char u[1..m], char v[1..n]):
    # for all i and j, d[i,j] will hold the Levenshtein distance between
    the first i characters of u and the first j characters of v, note that
    d has (m+1)*(n+1) values
    int d[0..m, 0..n]

    # set each element in d to zero
    d[0..m, 0..n] := 0

    # source prefixes can be transformed into empty string by dropping all
    characters
    for i from 1 to m:
       d[i, 0] := i

    # target prefixes can be reached from empty source prefix by inserting
    every character
    for j from 1 to n:
        d[0, j] := j

    # build the edit distance matrix
    for j from 1 to n:
        for i from 1 to m:
            if s[i] = t[j]:
                substitutionCost := 0
            else:
                substitutionCost := 1
                d[i, j] := minimum(d[i-1, j] + 1,                 # deletion
                                   d[i, j-1] + 1,                 # insertion
                                   d[i-1, j-1] + substitutionCost)  # substitution

    # the final value of the matrix is the edit distance between the terms
    return d[m, n]
```

You can see from the function definition pseudocode, how we have captured the necessary formulae we used earlier to define the Levenshtein Edit distance. We will now implement this pseudocode in Python. The algorithm uses O(mn) space, since it stores the entire distance matrix, but it is enough to just store the previous and current row of distances to get to the final result. We will do the same in our code, but we will also store the results in a matrix so that we can visualize them in the end. The following function implements the Levenshtein Edit distance.

```
import copy
import pandas as pd

def levenshtein_edit_distance(u, v):
    # convert to lower case
    u = u.lower()
    v = v.lower()
    # base cases
    if u == v: return 0
    elif len(u) == 0: return len(v)
    elif len(v) == 0: return len(u)
    # initialize edit distance matrix
    edit_matrix = []
    # initialize two distance matrices
    du = [0] * (len(v) + 1)
    dv = [0] * (len(v) + 1)
    # du: the previous row of edit distances
    for i in range(len(du)):
        du[i] = i
    # dv : the current row of edit distances
    for i in range(len(u)):
        dv[0] = i + 1
        # compute cost as per algorithm
        for j in range(len(v)):
            cost = 0 if u[i] == v[j] else 1
            dv[j + 1] = min(dv[j] + 1, du[j + 1] + 1, du[j] + cost)
        # assign dv to du for next iteration
        for j in range(len(du)):
            du[j] = dv[j]
        # copy dv to the edit matrix
        edit_matrix.append(copy.copy(dv))
    # compute the final edit distance and edit matrix
    distance = dv[len(v)]
    edit_matrix = np.array(edit_matrix)
    edit_matrix = edit_matrix.T
    edit_matrix = edit_matrix[1:,]
    edit_matrix = pd.DataFrame(data=edit_matrix,
                               index=list(v),
                               columns=list(u))
    return distance, edit_matrix
```

This function returns both the final Levenshtein Edit distance and the complete edit matrix between the two terms, **u** and **v**, which are taken as input. Remember we need to pass the terms directly in their raw string format and not their vector representations. Also we do not consider case of strings here and convert them to lowercase. The following snippet computes the Levenshtein Edit distance between our example terms using the previous function.

```
for term in other_terms:
    edit_d, edit_m = levenshtein_edit_distance(root_term, term)
    print('Computing distance between root: {} and term: {}'.format
    (root_term, term))
    print('Levenshtein edit distance is {}'.format(edit_d))
    print('The complete edit distance matrix is depicted below')
    print(edit_m)
    print('-'*30)
```

```
Computing distance between root: Believe and term: beleive
Levenshtein edit distance is 2
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
b  0  1  2  3  4  5  6
e  1  0  1  2  3  4  5
l  2  1  0  1  2  3  4
e  3  2  1  1  1  2  3
i  4  3  2  1  2  2  3
v  5  4  3  2  2  2  3
e  6  5  4  3  2  3  2
------------------------------
Computing distance between root: Believe and term: bargain
Levenshtein edit distance is 6
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
b  0  1  2  3  4  5  6
a  1  1  2  3  4  5  6
r  2  2  2  3  4  5  6
g  3  3  3  3  4  5  6
a  4  4  4  4  4  5  6
i  5  5  5  4  5  5  6
n  6  6  6  5  5  6  6
------------------------------
```

```
Computing distance between root: Believe and term: Elephant
Levenshtein edit distance is 7
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
e  1  1  2  3  4  5  6
l  2  2  1  2  3  4  5
e  3  2  2  2  2  3  4
p  4  3  3  3  3  3  4
h  5  4  4  4  4  4  4
a  6  5  5  5  5  5  5
n  7  6  6  6  6  6  6
t  8  7  7  7  7  7  7
------------------------------
```

You can see from this output that "Believe" and "beleive" are closest to each other with an edit distance of 2, and the distances between "Believe," "bargain," and "Elephant" are 6, indicating a total of six edit operations are needed. The edit distance matrices provide a more detailed insight into how the algorithm computes the distances per iteration.

Cosine Distance and Similarity

The *Cosine distance* is a metric that can be derived from the Cosine similarity and vice versa. Considering we have two terms represented in their vectorized forms (bag of character vectors that we shall depict shortly, whereby the order of the characters doesn't matter). Cosine similarity gives us the measure of the cosine of the angle between them when they are represented as non-zero positive vectors in an inner product space. Thus, term vectors that have a similar orientation will have scores closer to 1 (cos0°), indicating the vectors are very close to each other in the same direction (near to zero degree angle between them). Term vectors with a similarity score close to 0 (cos90°) indicate unrelated terms with a near orthogonal angle between then.
Term vectors with a similarity score close to -1 (cos180°) indicate terms that are completely oppositely oriented. Figure 7-2 illustrates this more clearly, where **u** and **v** are our term vectors in the vector space.

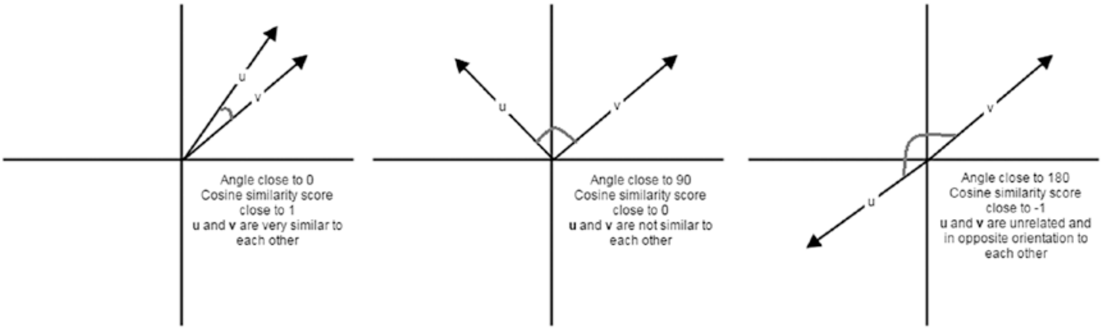


Figure 7-2. *Cosine similarity representations for term vectors*

Thus, you can see from the position of the vectors that the plots show more clearly how the vectors are close or far apart and the cosine of the angle between them gives us the Cosine similarity metric. Now we can formally define Cosine similarity as the dot product of the two term vectors, **u** and **v**, divided by the product of their L2 norms. Mathematically, we can represent the dot product between two vectors as follows:

$$u \cdot v = \| u \| \| v \| \cos(\theta)$$

where θ is the angle between **u** and **v** and $\|u\|$ represents the L2 norm for vector **u** and $\|v\|$ is the L2 norm for vector **v**. Thus, we can derive the Cosine similarity from the formula as follows:

$$cs(u,v) = \cos(\theta) = \frac{u \cdot v}{\| u \| \| v \|} = \frac{\sum_{i=1}^{n} u_i \, v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}}$$

where $cs(u, v)$ is the Cosine similarity score between **u** and ***v***. Here u_i and v_i are the various features of the two vectors and the total number of these features or components is **n**. In our case, we use the bag of characters vectorization to build these term vectors and **n** is the number of unique characters across the terms under analysis.

Bag of characters vectorization is very similar to the bag of words model except here we compute the frequency of each character in the word. Sequence or word orders are not taken into account here. The following function helps compute this.

```
from scipy.stats import itemfreq

def boc_term_vectors(word_list):
    word_list = [word.lower() for word in word_list]
    unique_chars = np.unique(
                        np.hstack([list(word)
                        for word in word_list]))
    word_list_term_counts = [{char: count
                              for char, count in np.stack(
                              np.unique(list(word), return_
                              counts=True), axis=1)}
                              for word in word_list]

    boc_vectors = [np.array([int(word_term_counts.get(char, 0))
                             for char in unique_chars])
                   for word_term_counts in word_list_term_counts]
    return list(unique_chars), boc_vectors
```

In this function, we take a list of words or terms and extract the unique characters from it. This becomes our feature list, just like we do with a bag of words. Instead of characters, unique words will be our features. Once we have this list of unique_chars we get the count for each character in each word and build our bag of characters vectors. The following code leverages the previous function to build the bag of character vectors for our sample terms.

```
# Bag of characters vectorization
import pandas as pd

feature_names, feature_vectors = boc_term_vectors(terms)
boc_df = pd.DataFrame(feature_vectors, columns=feature_names, index=terms)
print(boc_df)

          a  b  e  g  h  i  l  n  p  r  t  v
Believe   0  1  3  0  0  1  1  0  0  0  0  1
beleive   0  1  3  0  0  1  1  0  0  0  0  1
bargain   2  1  0  1  0  1  0  1  0  1  0  0
Elephant  1  0  2  0  1  0  1  1  1  0  1  0
```

Just like we expected, each vector is basically an unordered bag of characters depicting the frequency of each character in the corresponding word. Let's store these in specific variables before we compute cosine distances.

```
root_term_boc = boc_df[vec_df.index == root_term].values[0]
other_term_bocs = [boc_df[vec_df.index == term].values[0]
                      for term in other_terms]
```

An important point to note here is that the Cosine similarity score usually ranges from -1 to +1, but if we use the bag of characters based character frequencies for terms or bag of words based word frequencies for documents, the score will range from 0 to 1. This is because the frequency vectors can never be negative and hence the angle between the two vectors cannot exceed 90°. The Cosine distance is complementary to the similarity score and can be computed by the formula:

$$cd(u,v)=1-cs(u,v)=1-\cos(\theta)=1-\frac{u\cdot v}{\|u\|\|v\|}=1-\frac{\sum_{i=1}^{n}u_i\,v_i}{\sqrt{\sum_{i=1}^{n}u_i^2}\,\sqrt{\sum_{i=1}^{n}v_i^2}}$$

where $cd(u, v)$ denotes the Cosine distance between the term vectors **u** and **v**. The following function implements computation of Cosine distance based on the formulae.

```
def cosine_distance(u, v):
    distance = 1.0 - (np.dot(u, v) /
                        (np.sqrt(sum(np.square(u))) * np.sqrt(sum(np.square(v))))
                     )
    return distance
```

We now test the similarity between our example terms using their bag of character representations we created earlier and available in the `boc_root_vector` and the `boc_vector_terms` variables:

```
for term, boc_term in zip(other_terms, other_term_bocs):
    print('Analyzing similarity between root: {} and term: {}'.format
    (root_term, term))
    distance = round(cosine_distance(root_term_boc, boc_term), 2)
    similarity = round(1 - distance, 2)
    print('Cosine distance  is {}'.format(distance))
    print('Cosine similarity  is {}'.format(similarity))
    print('-'*40)
```

```
Analyzing similarity between root: Believe and term: beleive
Cosine distance  is -0.0
Cosine similarity  is 1.0
----------------------------------------
Analyzing similarity between root: Believe and term: bargain
Cosine distance  is 0.82
Cosine similarity  is 0.18
----------------------------------------
Analyzing similarity between root: Believe and term: Elephant
Cosine distance  is 0.39
Cosine similarity  is 0.61
----------------------------------------
```

These vector representations do not take the order of characters into account and hence the similarity between the terms "Believe" and "beleive" is 1.0 or a perfect 100%. You can see how this can be used in combination with a semantic dictionary like WordNet to provide correct spelling suggestions. It does this by suggesting semantically and syntactically correct words from a vocabulary when users misspell a word by measuring the similarity between the words.

You can even try out different features here instead of single character frequencies, like taking two characters at a time and computing their frequencies to build the term vectors. This will take into account some of the sequences that characters maintain in various terms. Try out different possibilities and compare the results! This distance measure works really well when measuring similarity between large documents or sentences and we will see that in the following section when we discuss document similarity.

Analyzing Document Similarity

We analyzed similarity between terms using various similarity and distance metrics in the previous sections. We also saw how vectorization was useful so that mathematical computations become much easier, especially when computing distances between vectors. In this section, we try to analyze similarities between documents. By now, you must already know that a document is defined as a body of text, which can comprise sentences or paragraphs of text. For analyzing document similarity, we will be doing some basic text preprocessing, vectorizing documents using the TF-IDF scheme, similar to what we did previously when we classified text documents or summarized entire documents. Once we have the vector representations of the various documents, we can compute similarity between the documents using some standard distance or similarity metrics. The metrics we cover in this section are as follows:

- Cosine similarity
- Okapi BM25 ranking

Just like usual, we cover the concepts behind each metric, look at the mathematical representations and definitions, and then implement them using Python. To make things interesting, we actually showcase these with a real-world example of trying to build a movie recommender system! Consider this a mini-simulation of what happens when you search for or watch a movie online and similar movies are recommended to you based on the movie description and content. In the real world, you obviously will have more parameters and features like ratings, genre, user preference history, and so on, but the recommender we build will actually showcase how document similarity can help build simple but amazing recommenders.

Building a Movie Recommender

Recommender systems are one of the popular and most adopted applications of machine learning. They are typically used to recommend entities to users. These entities can be anything like products, movies, services, and so on. Popular examples of recommendations include:

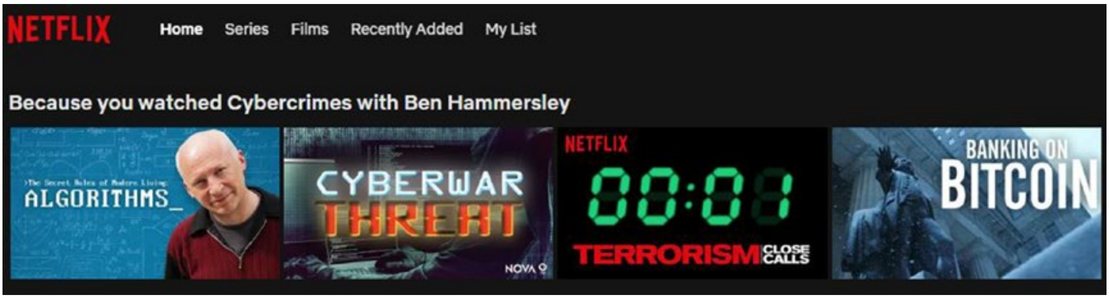
- Amazon suggesting products on its website
- Amazon Prime, Netflix, and Hotstar recommending movies/shows
- YouTube recommending videos to watch

Recommender systems can typically be implemented in three ways:

- **Simple rule-based recommenders:** Based on specific global metrics and thresholds like movie popularity, global ratings, etc.
- **Content-based recommenders:** Based on providing similar entities based on a specific entity of interest. Content metadata can be used here, such as movie description, genre, cast, director, and so on.

- **Collaborative filtering recommenders:** We don't need metadata but we try to predict recommendations and ratings based on past ratings of different users and specific items.

We build a movie recommendation system whereby, based on data/metadata pertaining to different movies, we try to recommend similar movies of interest. See Figure 7-3.



Because you watched Cybercrimes with Ben Hammersley

Figure 7-3. *Typical movie or TV show recommendations*

Since our focus in not on recommendation engines but on NLP, we leverage the text-based metadata for each movie to try to recommend similar movies based on specific movies of interest. This falls under content-based recommenders. We follow a step-by-step approach to building this recommender system with document similarity.

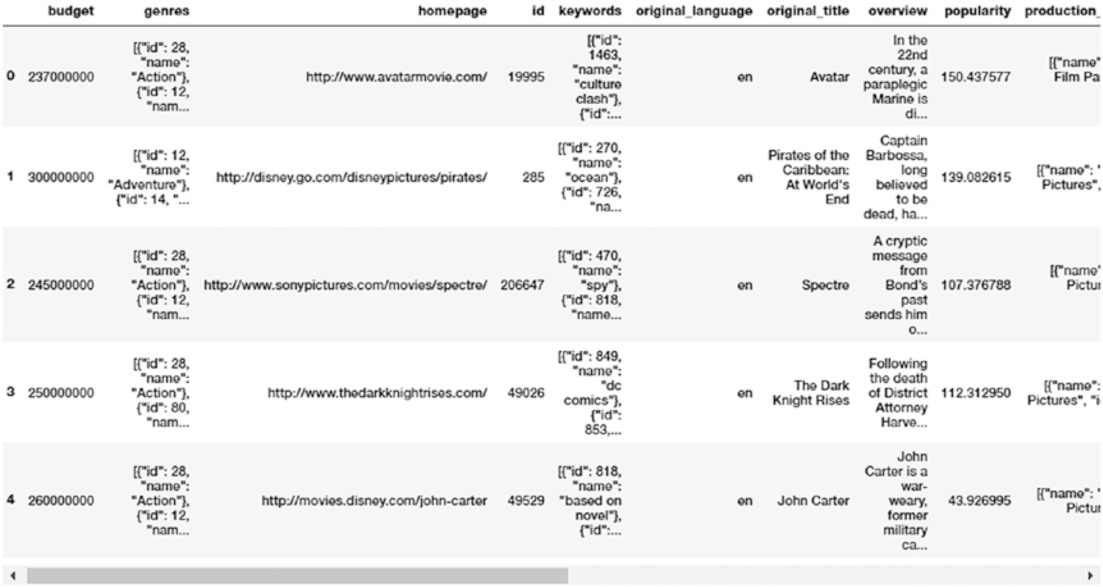
Load and View Dataset

We will be using the very popular TMDB 5,000 movies dataset for this experiment, which you can find on Kaggle at `https://www.kaggle.com/tmdb/tmdb-movie-metadata/home`. But we will also be providing a nice compressed version of the same dataset in our official GitHub repository, which you can obtain from `https://github.com/dipanjanS/text-analytics-with-python`. Let's load and view this dataset now. See Figure 7-4.

```
import pandas as pd

df = pd.read_csv('tmdb_5000_movies.csv.gz', compression='gzip')
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4803 entries, 0 to 4802
```

```
Data columns (total 20 columns):
budget                  4803 non-null int64
genres                  4803 non-null object
homepage                1712 non-null object
id                      4803 non-null int64
keywords                4803 non-null object
original_language       4803 non-null object
original_title          4803 non-null object
overview                4800 non-null object
popularity              4803 non-null float64
production_companies    4803 non-null object
production_countries    4803 non-null object
release_date            4802 non-null object
revenue                 4803 non-null int64
runtime                 4801 non-null float64
spoken_languages        4803 non-null object
status                  4803 non-null object
tagline                 3959 non-null object
title                   4803 non-null object
vote_average            4803 non-null float64
vote_count              4803 non-null int64
dtypes: float64(3), int64(4), object(13)
memory usage: 750.5+ KB
```

```
df.head()
```



	budget	genres	homepage	id	keywords	original_language	original_title	overview	popularity	production_
0	237000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.avatarmovie.com/	19995	[{"id": 1463, "name": "culture clash"}, {"id":...	en	Avatar	In the 22nd century, a paraplegic Marine is di...	150.437577	[{"name" Film Pa
1	300000000	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	http://disney.go.com/disneypictures/pirates/	285	[{"id": 270, "name": "ocean"}, {"id": 726, "na...	en	Pirates of the Caribbean: At World's End	Captain Barbossa, long believed to be dead, ha...	139.082615	[{"name": ' Pictures",
2	245000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://www.sonypictures.com/movies/spectre/	206647	[{"id": 470, "name": "spy"}, {"id": 818, "name...	en	Spectre	A cryptic message from Bond's past sends him o...	107.376788	[{"name' Pictu
3	250000000	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	http://www.thedarkknightrises.com/	49026	[{"id": 849, "name": "dc comics"}, {"id": 853,...	en	The Dark Knight Rises	Following the death of District Attorney Harve...	112.312950	[{"name": Pictures", "i
4	260000000	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	http://movies.disney.com/john-carter	49529	[{"id": 818, "name": "based on novel"}, {"id":...	en	John Carter	John Carter is a war-weary, former military ca...	43.926995	[{"name": ' Pictu

Figure 7-4. *The TMDB 5,000 movies dataset*

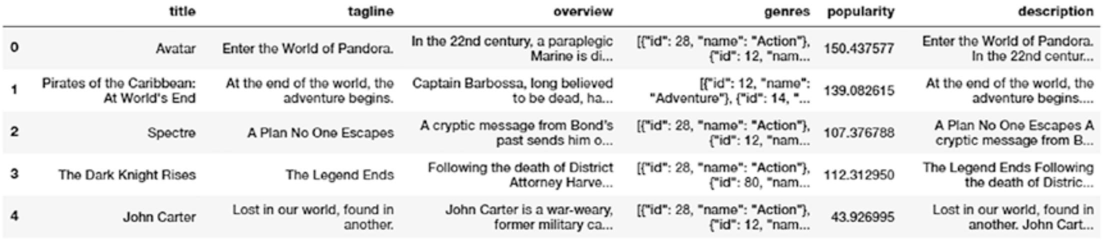
Obviously for our simple content-based document similarity movie recommender, we do not need all these fields for our analysis (although they might be useful if you want to build a more sophisticated system). We will also combine the text content from the movie tagline and overview columns into a new column called `description`. See Figure 7-5.

```
df = df[['title', 'tagline', 'overview', 'genres', 'popularity']]
df.tagline.fillna(", inplace=True)
df['description'] = df['tagline'].map(str) + ' ' + df['overview']
df.dropna(inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4800 entries, 0 to 4802
Data columns (total 6 columns):
title          4800 non-null object
tagline        4800 non-null object
overview       4800 non-null object
genres         4800 non-null object
popularity     4800 non-null float64
description    4800 non-null object
```

```
dtypes: float64(1), object(5)
memory usage: 262.5+ KB
```

```
df.head()
```



	title	tagline	overview	genres	popularity	description
0	Avatar	Enter the World of Pandora.	In the 22nd century, a paraplegic Marine is di...	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	150.437577	Enter the World of Pandora. In the 22nd centur...
1	Pirates of the Caribbean: At World's End	At the end of the world, the adventure begins.	Captain Barbossa, long believed to be dead, ha...	[{"id": 12, "name": "Adventure"}, {"id": 14, "...	139.082615	At the end of the world, the adventure begins....
2	Spectre	A Plan No One Escapes	A cryptic message from Bond's past sends him o...	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	107.376788	A Plan No One Escapes A cryptic message from B...
3	The Dark Knight Rises	The Legend Ends	Following the death of District Attorney Harve...	[{"id": 28, "name": "Action"}, {"id": 80, "nam...	112.312950	The Legend Ends Following the death of Distric...
4	John Carter	Lost in our world, found in another.	John Carter is a war-weary, former military ca...	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	43.926995	Lost in our world, found in another. John Cart...

Figure 7-5. *The TMDB 5,000 movies dataset with relevant attributes*

Now, we will build our own movie recommender system. The major components going into its pipeline are as follows:

- Text preprocessing
- Feature engineering
- Document similarity computation
- Find top similar movies based on a sample movie
- Build a movie recommender

Recommendations are all about understanding the underlying features that make us favor one choice over the other. Similarity between items (in this case, movies) is one way to understand why we choose one movie over another. There are different ways to calculate similarity between two items. One of the most widely used measures is Cosine similarity, which we used earlier and we will be using again shortly.

Text Preprocessing

We will do some basic text preprocessing on our movie descriptions before we build our features. Nothing too fancy, since the intent here is to focus on document similarity and not on text processing.

```
import nltk
import re
import numpy as np
```

```
stop_words = nltk.corpus.stopwords.words('english')

def normalize_document(doc):
    # lower case and remove special characters\whitespaces
    doc = re.sub(r'[^a-zA-Z0-9\s]', ", doc, re.I|re.A)
    doc = doc.lower()
    doc = doc.strip()
    # tokenize document
    tokens = nltk.word_tokenize(doc)
    # filter stopwords out of document
    filtered_tokens = [token for token in tokens if token not in stop_words]
    # re-create document from filtered tokens
    doc = ' '.join(filtered_tokens)
    return doc

normalize_corpus = np.vectorize(normalize_document)

norm_corpus = normalize_corpus(list(df['description']))
len(norm_corpus)
```

```
4800
```

Let's move on to text representation, which can be done by leveraging some feature engineering scheme like TF-IDF.

Extract TF-IDF Features

We talked about the TF-IDF representation scheme in extensive detail in Chapter 4. Here, we leverage it to vectorize our preprocessed movie descriptions, thereby converting them into numeric vectors.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tf = TfidfVectorizer(ngram_range=(1, 2), min_df=2)
tfidf_matrix = tf.fit_transform(norm_corpus)
tfidf_matrix.shape
```

```
(4800, 20667)
```

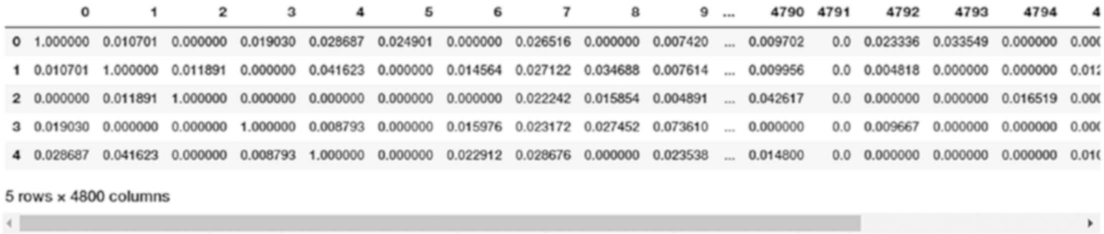
We take uni-gram and bi-grams as our features and remove terms that occur only in one document across the whole corpus. Now that we have our documents normalized and vectorized with `tf-idf`-based vector representations, we look at how to compute document similarity with cosine similarity.

Cosine Similarity for Pairwise Document Similarity

We have seen the concepts with regard to computing Cosine similarity and implemented them for term similarity. Here, we reuse the same concepts to compute the Cosine similarity scores for documents instead of terms. The document vectors will be the bag of words model-based vectors with TF-IDF values instead of term frequencies. Thus, we should end up getting an *N x N* matrix where *N* is equal to the number of movies, which is 4,800. See Figure 7-6.

```
from sklearn.metrics.pairwise import cosine_similarity
```

```
doc_sim = cosine_similarity(tfidf_matrix)
doc_sim_df = pd.DataFrame(doc_sim)
doc_sim_df.head()
```



	0	1	2	3	4	5	6	7	8	9	...	4790	4791	4792	4793	4794	4
0	1.000000	0.010701	0.000000	0.019030	0.028687	0.024901	0.000000	0.026516	0.000000	0.007420	...	0.009702	0.0	0.023336	0.033549	0.000000	0.00
1	0.010701	1.000000	0.011891	0.000000	0.041623	0.000000	0.014564	0.027122	0.034688	0.007614	...	0.009956	0.0	0.004818	0.000000	0.000000	0.01
2	0.000000	0.011891	1.000000	0.000000	0.000000	0.000000	0.000000	0.022242	0.015854	0.004891	...	0.042617	0.0	0.000000	0.000000	0.016519	0.00
3	0.019030	0.000000	0.000000	1.000000	0.008793	0.000000	0.015976	0.023172	0.027452	0.073610	...	0.000000	0.0	0.009667	0.000000	0.000000	0.00
4	0.028687	0.041623	0.000000	0.008793	1.000000	0.000000	0.022912	0.028676	0.000000	0.023538	...	0.014800	0.0	0.000000	0.000000	0.000000	0.01

5 rows × 4800 columns

__Figure 7-6.__ Pairwise document cosine similarity

Now, we build a workflow to determine the most similar and recommended movies for a sample movie before building our movie recommender system. Before we do this, let's build a list of all movie titles in our dataset.

```
movies_list = df['title'].values
movies_list, movies_list.shape
```

```
(array(['Avatar', "Pirates of the Caribbean: At World's End", 'Spectre',
        ..., 'Signed, Sealed, Delivered', 'Shanghai Calling',
        'My Date with Drew'], dtype=object), (4800,))
```

We can typically index into our pairwise similarity matrix to get document similarities for the recommendations.

Find Top Similar Movies for a Sample Movie

Let's take *Minions*, one of the most popular movies, and try to find the most similar movies to recommend. The following are the major steps that will be helpful to us later to build a generic function.

Find Movie ID

Since we have a list of movies, finding the position index of the movie in our dataset is pretty straightforward.

```
movie_idx = np.where(movies_list == 'Minions')[0][0]
movie_idx
```

```
546
```

Get Movie Similarities

We will now use this positional index to obtain the vector of pairwise movie similarities for all movies with the movie *Minions* having an index 546.

```
movie_similarities = doc_sim_df.iloc[movie_idx].values
movie_similarities
```

```
array([0.0104544 , 0.01072835, 0.        , ..., 0.00690954, 0.        ,
       0.        ])
```

Get Top Five Similar Movie IDs

It is now time to get the top five movies that are the most similar to the movie *Minions*. Remember that we are not interested in showing the similarity values, but in getting the movie indices.

```
similar_movie_idxs = np.argsort(-movie_similarities)[1:6]
similar_movie_idxs
```

```
array([506, 614, 241, 813, 154], dtype=int64)
```

Get Top Five Similar Movies

We can easily obtain the top five similar movies to *Minions* since we already have the movie index positions.

```
similar_movies = movies_list[similar_movie_idxs]
similar_movies
```

```
array(['Despicable Me 2', 'Despicable Me',
       'Teenage Mutant Ninja Turtles: Out of the Shadows', 'Superman',
       'Rise of the Guardians'], dtype=object)
```

Not bad! The top two movies are definitely very similar to *Minions* and are in fact all a part of the *Despicable Me* franchise.

Build a Movie Recommender

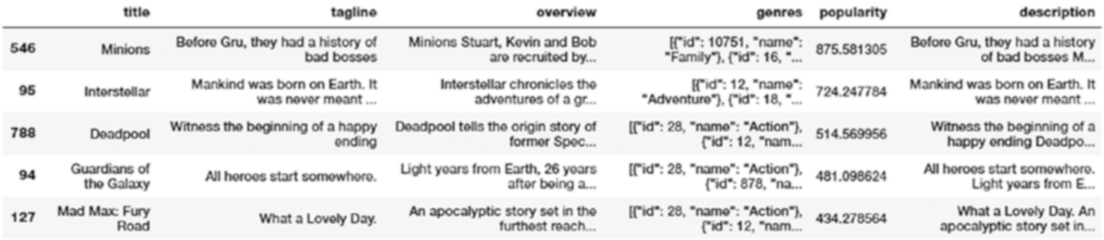
It's time now to put together everything we have learned and build our movie recommender. We will build a movie recommender function to recommend movies. This function will require the movie title, movie title list, and document similarity matrix dataframe as inputs to the function.

```
def movie_recommender(movie_title, movies=movies_list, doc_sims=doc_sim_df):
    # find movie id
    movie_idx = np.where(movies == movie_title)[0][0]
    # get movie similarities
    movie_similarities = doc_sims.iloc[movie_idx].values
    # get top 5 similar movie IDs
    similar_movie_idxs = np.argsort(-movie_similarities)[1:6]
    # get top 5 movies
    similar_movies = movies[similar_movie_idxs]
    # return the top 5 movies
    return similar_movies
```

Get a List of Popular Movies

We can sort our movies dataset based on popularity score and select some of the most popular movies. We can then view their recommendations for some interesting results! See Figure 7-7.

```
pop_movies = df.sort_values(by='popularity', ascending=False)
pop_movies.head()
```



	title	tagline	overview	genres	popularity	description
546	Minions	Before Gru, they had a history of bad bosses	Minions Stuart, Kevin and Bob are recruited by...	[{"id": 10751, "name": "Family"}, {"id": 16, "...	875.581305	Before Gru, they had a history of bad bosses M...
95	Interstellar	Mankind was born on Earth. It was never meant ...	Interstellar chronicles the adventures of a gr...	[{"id": 12, "name": "Adventure"}, {"id": 18, "...	724.247784	Mankind was born on Earth. It was never meant ...
788	Deadpool	Witness the beginning of a happy ending	Deadpool tells the origin story of former Spec...	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	514.569956	Witness the beginning of a happy ending Deadpo...
94	Guardians of the Galaxy	All heroes start somewhere.	Light years from Earth, 26 years after being a...	[{"id": 28, "name": "Action"}, {"id": 878, "na...	481.098624	All heroes start somewhere. Light years from E...
127	Mad Max: Fury Road	What a Lovely Day.	An apocalyptic story set in the furthest reach...	[{"id": 28, "name": "Action"}, {"id": 12, "nam...	434.278564	What a Lovely Day. An apocalyptic story set in...

Figure 7-7. *Popular movies*

I selected the following movies based on their popularity score and how interesting they might be. Feel free to substitute them with your own movies.

```
popular_movies = ['Minions', 'Interstellar', 'Deadpool', 'Jurassic World',
                  'Pirates of the Caribbean: The Curse of the Black Pearl',
                  'Dawn of the Planet of the Apes', 'The Hunger Games:
                  Mockingjay - Part 1', 'Terminator Genisys', 'Captain
                  America: Civil War', 'The Dark Knight', 'The Martian',
                  'Batman v Superman: Dawn of Justice', 'Pulp Fiction', 'The
                  Godfather', 'The Shawshank Redemption', 'The Lord of the
                  Rings: The Fellowship of the Ring', 'Harry Potter and the
                  Chamber of Secrets', 'Star Wars', 'The Hobbit: The Battle
                  of the Five Armies', 'Iron Man']
```

Let's get the top five recommended movies for each of these movies using our movie recommender function now.

```
for movie in popular_movies:
    print('Movie:', movie)
    print('Top 5 recommended Movies:', movie_recommender(movie_
    title=movie))
    print()
```

```
Movie: Minions
Top 5 recommended Movies: ['Despicable Me 2' 'Despicable Me'
 'Teenage Mutant Ninja Turtles: Out of the Shadows' 'Superman'
 'Rise of the Guardians']

...
...

Movie: Jurassic World
Top 5 recommended Movies: ['Jurassic Park' 'The Lost World: Jurassic Park'
'The Nut Job'
 "National Lampoon's Vacation" 'Vacation']

Movie: Pirates of the Caribbean: The Curse of the Black Pearl
Top 5 recommended Movies: ["Pirates of the Caribbean: Dead Man's Chest"
'The Pirate'
 'Pirates of the Caribbean: On Stranger Tides'
 'The Pirates! In an Adventure with Scientists!' 'Joyful Noise']
...
...

Movie: Captain America: Civil War
Top 5 recommended Movies: ['Captain America: The Winter Soldier' 'This
Means War'
 'Avengers: Age of Ultron' 'Iron Man 2' 'Escape from Tomorrow']

Movie: The Dark Knight
Top 5 recommended Movies: ['The Dark Knight Rises' 'Batman Forever' 'Batman
Returns'
 'Batman: The Dark Knight Returns, Part 2' 'Slow Burn']

Movie: The Martian
Top 5 recommended Movies: ['The Last Days on Mars' 'Swept Away' 'Alive'
'All Is Lost' 'Red Planet']

...
...
```

```
Movie: The Lord of the Rings: The Fellowship of the Ring
Top 5 recommended Movies: ['The Lord of the Rings: The Two Towers'
 'The Hobbit: The Desolation of Smaug'
 'The Lord of the Rings: The Return of the King'
 "What's the Worst That Could Happen?" 'The Hobbit: An Unexpected Journey']

Movie: Harry Potter and the Chamber of Secrets
Top 5 recommended Movies: ['Harry Potter and the Prisoner of Azkaban'
 'Harry Potter and the Goblet of Fire'
 'Harry Potter and the Order of the Phoenix'
 'Harry Potter and the Half-Blood Prince'
 "Harry Potter and the Philosopher's Stone"]

Movie: Star Wars
Top 5 recommended Movies: ['The Empire Strikes Back' 'Return of the Jedi'
'Shrek the Third'
 'The Ice Pirates' 'The Tale of Despereaux']
Movie: The Hobbit: The Battle of the Five Armies
Top 5 recommended Movies: ['The Hobbit: The Desolation of Smaug' 'The
Hobbit: An Unexpected Journey'
 "Dragon Nest: Warriors' Dawn"
 'A Funny Thing Happened on the Way to the Forum' 'X-Men: Apocalypse']

Movie: Iron Man
Top 5 recommended Movies: ['Iron Man 2' 'Avengers: Age of Ultron' 'Hostage'
'Iron Man 3'
 'Baahubali: The Beginning']
```

Based on the results, you can clearly see our simple document similarity based recommender is performing really well! We recommend using Scikit-Learn's `cosine_similarity()` utility function, which is quite useful. You can also use Gensim's `similarities` module or the `cossim()` function directly, available in the `gensim.matutils` module.

Okapi BM25 Ranking for Pairwise Document Similarity

There are several techniques that are quite popular in information retrieval and search engines, including PageRank and Okapi BM25. The term BM stands for *best matching*. This technique is also known just as BM25, but for the sake of completeness, we refer to it as Okapi BM25, because while the concepts behind the BM25 function were originally theoretical, the City University in London built the Okapi Information Retrieval system in the 1980-90s, which implemented this technique to retrieve documents on actual real-world data.

This technique can also be called a framework or model based on probabilistic relevancy and was developed by several people in the 1970-80s, including computer scientists S. Robertson and K. Jones. There are several functions that rank documents based on different factors and BM25 is one of them. Its newer variant is BM25F and some other variants include BM15 and BM25+.

The Okapi BM25 technique can be formally defined as a document ranking and retrieval function based on a bag of words-based model for retrieving relevant documents based on a user input query. This query can be a document containing a sentence or collection of sentences or it can even be a couple of words. Okapi BM25 is actually not just a single function but is a framework consisting of a whole collection of scoring functions that are combined.

Consider we have a query document **QD** such that $QD = (q_1, q_2, \ldots, q_n)$ containing **n** terms or keywords and we have a corpus document **CD** in the corpus of documents from which we want to get the most relevant documents to the query document based on similarity scores. Assuming we have these, we can mathematically define the BM25 score between these two documents as follows:

$$bm25(CD,QD) = \sum_{i=1}^{n} idf(q_i) \cdot \frac{f(q_i,\ CD) \cdot (k_1 + 1)}{f(q_i,\ CD) + k_1 \cdot (1 - b + b \cdot \frac{|CD|}{avgdl}}$$

where the function $bm25(CD, QD)$ computes the BM25 rank or score of the document **CD**, based on the query document **QD**. The function $idf(q_i)$ gives us the inverse document frequency (***idf***) of the term q_i in the corpus, which contains **CD** and from which we want to retrieve the relevant documents. If you remember, we computed **idfs** in Chapter 4 when we implemented the `tf-idf` feature extractor. Just to refresh your memory, it can represented by

$$idf(t) = 1 + \log \frac{C}{1 + df(t)}$$

where $idf(t)$ represents the **idf** for the term **t**, **C** represents the count of the total number of documents in our corpus, and $df(t)$ represents the number of documents in which the term **t** is present. There are various other methods of implementing **idf**, but we will be using this one.

On a side note, the end outcome from the different implementations is very similar. The function $f(q_i, CD)$ gives us the frequency of the term q_i in the corpus document **CD**. The expression $|CD|$ indicates the total length of the document **CD***,* which is measured by the number of words and the term *avgdl* represents the average document length of the corpus from which we will be retrieving documents. Besides there, you will also observe there are two free parameters—k_1 is usually in the range of [1.2, 2.0] and b is usually taken as 0.75. We will be taking the value of k_1 to be 2.5 in our implementation and b to be 0.85, based on the popular implementation of the BM25 algorithm in the Gensim framework.

There are several steps that we must go through to successfully implement and compute BM25 scores for documents:

1. Calculate frequencies of terms in documents and in corpus.
2. Compute the inverse document frequencies of terms.
3. Get bag of words-based features for corpus documents and query documents.
4. Build a function to compute the BM25 score of a given document in relation to a specific document from the corpus.
5. Build a function that leverages the function from Step 4, which computes and returns BM25 scores of a given document in relation to every other document in the corpus (like a vector of similarities for each document).
6. Build a function that returns pairwise BM25 similarity scores (weights) for all the documents in the corpus (leverages the function from Step 5).

The code we implement here has actually been adopted from the Gensim framework and we definitely recommend it if you are interested in leveraging BM25 similarity. We show the internals of the similarity framework so you can correlate it with the earlier defined concepts. The following class helps implement all the components from Steps 1 through 5 in our defined workflow.

```
"""
Data:
-----
.. data:: PARAM_K1 - Free smoothing parameter for BM25.
.. data:: PARAM_B - Free smoothing parameter for BM25.
.. data:: EPSILON - Constant used for negative idf of document in corpus.
"""

import math
from six import iteritems
from six.moves import xrange

PARAM_K1 = 2.5
PARAM_B = 0.85
EPSILON = 0.2

class BM25(object):
    """Implementation of Best Matching 25 ranking function.
    Attributes
    ----------
    corpus_size : int
        Size of corpus (number of documents).
    avgdl : float
        Average length of document in `corpus`.
    corpus : list of list of str
        Corpus of documents.
    f : list of dicts of int
        Dictionary with terms frequencies for each document in `corpus`.
        Words used as keys and frequencies as values.
    df : dict
        Dictionary with terms frequencies for whole `corpus`.
        Words used as keys and frequencies as values.
```

```
idf : dict
    Dictionary with inversed terms frequencies for whole `corpus`.
    Words used as keys and frequencies as values.
doc_len : list of int
    List of document lengths.
"""
def __init__(self, corpus):
    """
    Parameters
    ----------
    corpus : list of list of str
        Given corpus.
    """
    self.corpus_size = len(corpus)
    self.avgdl = sum(float(len(x)) for x in corpus) / self.corpus_size
    self.corpus = corpus
    self.f = []
    self.df = {}
    self.idf = {}
    self.doc_len = []
    self.initialize()

def initialize(self):
    """Calculates frequencies of terms in documents and in corpus.
       Also computes inverse document frequencies."""
    for document in self.corpus:
        frequencies = {}
        self.doc_len.append(len(document))

        for word in document:
            if word not in frequencies:
                frequencies[word] = 0
            frequencies[word] += 1
        self.f.append(frequencies)
```

```
            for word, freq in iteritems(frequencies):
                if word not in self.df:
                    self.df[word] = 0
                self.df[word] += 1

        for word, freq in iteritems(self.df):
            self.idf[word] = math.log(self.corpus_size - freq + 0.5) -
            math.log(freq + 0.5)

    def get_score(self, document, index, average_idf):
        """Computes BM25 score of given `document` in relation to item of
        corpus
           selected by `index`.
        Parameters
        ----------
        document : list of str
            Document to be scored.
        index : int
            Index of document in corpus selected to score with `document`.
        average_idf : float
            Average idf in corpus.
        Returns
        -------
        float
            BM25 score.
        """
        score = 0
        for word in document:
            if word not in self.f[index]:
                continue
            idf = self.idf[word] if self.idf[word] >= 0 else EPSILON *
            average_idf
            score += (idf * self.f[index][word] * (PARAM_K1 + 1)
                      / (self.f[index][word] + PARAM_K1 * (1 - PARAM_B +
                      PARAM_B * self.doc_len[index] / self.avgdl)))
        return score
    def get_scores(self, document, average_idf):
        """Computes and returns BM25 scores of given `document` in
        relation to every item in corpus.
        Parameters
        ----------
        document : list of str
            Document to be scored.
        average_idf : float
            Average idf in corpus.
        Returns
        -------
        list of float
            BM25 scores.
        """
        scores = []
        for index in xrange(self.corpus_size):
            score = self.get_score(document, index, average_idf)
            scores.append(score)
        return scores
```

We can now implement the function from Step 6 in our workflow to compute pairwise document BM25 similarity scores, which is exactly what we need!

```
def get_bm25_weights(corpus):
    """Returns BM25 scores (weights) of documents in corpus.
       Each document has to be weighted with every document in given corpus.
    Parameters
    ----------
    corpus : list of list of str
        Corpus of documents.
    Returns
    -------
    list of list of float
        BM25 scores.
    Examples
    --------
    >>> from gensim.summarization.bm25 import get_bm25_weights
    >>> corpus = [
    ...     ["black", "cat", "white", "cat"],
    ...     ["cat", "outer", "space"],
    ...     ["wag", "dog"]
    ... ]
    >>> result = get_bm25_weights(corpus)
    """
    bm25 = BM25(corpus)
    average_idf = sum(float(val) for val in bm25.idf.values()) / len(bm25.idf)

    weights = []
    for doc in corpus:
        scores = bm25.get_scores(doc, average_idf)
        weights.append(scores)

    return weights
```

To use this function based on the documentation, we need to tokenize our corpus first, as depicted in the following code.

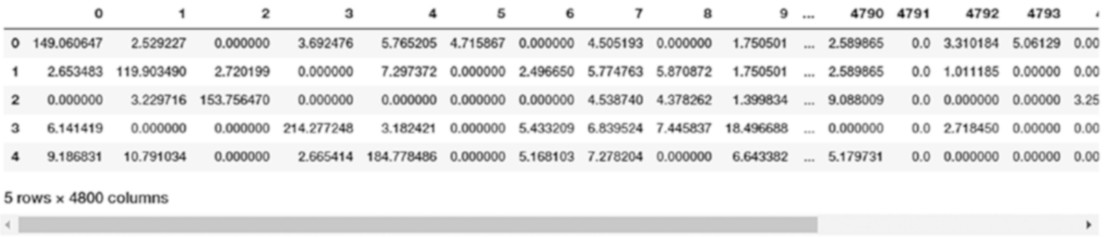
```
norm_corpus_tokens = np.array([nltk.word_tokenize(doc) for doc in norm_
corpus])
norm_corpus_tokens[:3]

array([list(['enter', 'world', 'pandora', '22nd', 'century', 'paraplegic',
             'marine', 'dispatched', 'moon', 'pandora', 'unique',
             'mission', 'becomes', 'torn', 'following', 'orders',
             'protecting', 'alien', 'civilization']),
       list(['end', 'world', 'adventure', 'begins', 'captain', 'barbossa',
             'long', 'believed', 'dead', 'come', 'back', 'life', 'headed',
             'edge', 'earth', 'turner', 'elizabeth', 'swann', 'nothing',
             'quite', 'seems']),
list(['plan', 'one', 'escapes', 'cryptic', 'message', 'bonds',
     'past', 'sends', 'trail', 'uncover', 'sinister',
     'organization', 'battles', 'political', 'forces', 'keep',
     'secret', 'service', 'alive', 'bond', 'peels', 'back',
     'layers', 'deceit', 'reveal', 'terrible', 'truth', 'behind',
     'spectre'])], dtype=object)
```

We can now use our previously defined get_bm25_weights(…) function to build the pairwise document similarity matrix. Remember that this takes a fair bit of time to compute, depending on the corpus size. See Figure 7-8.

```
%%time
wts = get_bm25_weights(norm_corpus_tokens)
```

```
Wall time: 2min 28s
```

```
# viewing our pairwise similarity matrix
bm25_wts_df = pd.DataFrame(wts)
bm25_wts_df.head()
```



	0	1	2	3	4	5	6	7	8	9	...	4790	4791	4792	4793	
0	149.060647	2.529227	0.000000	3.692476	5.765205	4.715867	0.000000	4.505193	0.000000	1.750501	...	2.589865	0.0	3.310184	5.06129	0.00
1	2.653483	119.903490	2.720199	0.000000	7.297372	0.000000	2.496650	5.774763	5.870872	1.750501	...	2.589865	0.0	1.011185	0.00000	0.00
2	0.000000	3.229716	153.756470	0.000000	0.000000	0.000000	0.000000	4.538740	4.378262	1.399834	...	9.088009	0.0	0.000000	0.00000	3.25
3	6.141419	0.000000	0.000000	214.277248	3.182421	0.000000	5.433209	6.839524	7.445837	18.496688	...	0.000000	0.0	2.718450	0.00000	0.00
4	9.186831	10.791034	0.000000	2.665414	184.778486	0.000000	5.168103	7.278204	0.000000	6.643382	...	5.179731	0.0	0.000000	0.00000	0.00

5 rows × 4800 columns

Figure 7-8. Our pairwise BM25 document similarity matrix

We can now use our movie recommender function to get the top five movie recommendations for the popular movies we selected earlier.

```
for movie in popular_movies:
    print('Movie:', movie)
    print('Top 5 recommended Movies:', movie_recommender(movie_title=movie,
    doc_sims=bm25_wts_df))
    print()
```

```
Movie: Minions
Top 5 recommended Movies: ['Despicable Me 2' 'Despicable Me'
 'Teenage Mutant Ninja Turtles: Out of the Shadows' 'Intolerance'
 'Superman']

Movie: Interstellar
Top 5 recommended Movies: ['Space Pirate Captain Harlock' 'Prometheus'
'Starship Troopers' 'Gattaca'
 'Space Cowboys']

...
...

Movie: The Lord of the Rings: The Fellowship of the Ring
Top 5 recommended Movies: ['The Lord of the Rings: The Two Towers'
 'The Lord of the Rings: The Return of the King'
 'The Hobbit: The Desolation of Smaug' 'The Hobbit: An Unexpected Journey'
 "What's the Worst That Could Happen?"]

Movie: Harry Potter and the Chamber of Secrets
Top 5 recommended Movies: ['Harry Potter and the Goblet of Fire'
 'Harry Potter and the Prisoner of Azkaban'
 'Harry Potter and the Half-Blood Prince'
 'Harry Potter and the Order of the Phoenix'
 "Harry Potter and the Philosopher's Stone"]

Movie: Star Wars
Top 5 recommended Movies: ['The Empire Strikes Back' 'Return of the Jedi'
'Shanghai Noon'
 'The Ice Pirates' 'The Tale of Despereaux']
Movie: The Hobbit: The Battle of the Five Armies
Top 5 recommended Movies: ['The Hobbit: The Desolation of Smaug' 'The
Hobbit: An Unexpected Journey'
 "Dragon Nest: Warriors' Dawn" 'Harry Potter and the Order of the Phoenix'
 '300: Rise of an Empire']
```

```
Movie: Iron Man
Top 5 recommended Movies: ['Iron Man 2' 'Avengers: Age of Ultron' 'Iron Man 3'
'Batman Begins'
 'Street Fighter']
```

We recommend using Gensim's `bm25` module under the `gensim.summarization` module. If you are interested, you should definitely give it a try.

Try loading a bigger corpus of documents and testing these functions on some sample query strings and documents. In fact, information retrieval frameworks like Solr and ElasticSearch are built on top of Lucene, which use these types of ranking algorithms to return relevant documents from an index of stored documents. You can build your own search engine using them! Interested readers can check out this link `https://www.elastic.co/blog/found-bm-vs-lucene-default-similarity` by `elastic.co`, which is the company behind the popular ElasticSearch product. The performance of BM25 is much better than the default similarity ranking implementation of Lucene.

Document Clustering

Document clustering or *cluster analysis* is an interesting area in natural language processing and text analytics that applies unsupervised machine learning concepts and techniques. The main premise of document clustering is similar to that of document categorization, whereby you start with a whole corpus of documents and you are tasked with segregating them into various groups based on some distinctive properties, attributes, and features of the documents. Document classification needs labeled training data to build a model and then categorize documents. Document clustering uses unsupervised machine learning algorithms to group the documents into various clusters. The properties of these clusters are such that documents inside one cluster are more similar and related to each other as compared to documents belonging to other clusters. Figure 7-9, courtesy of Scikit-Learn, visualizes an example of clustering data points into three clusters based on its features.

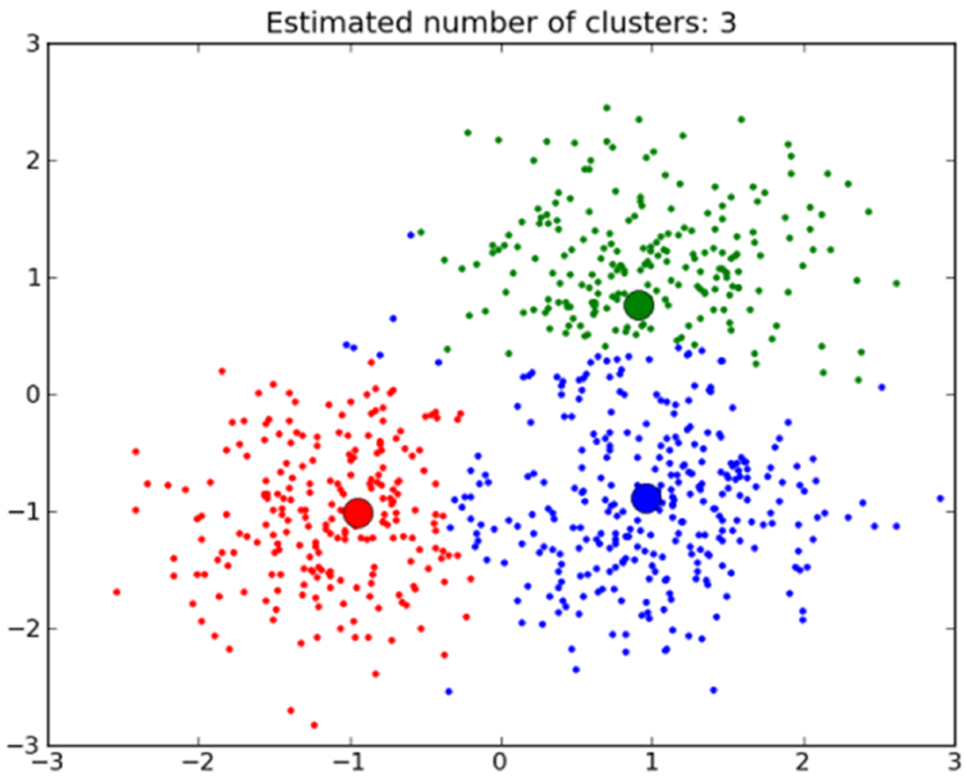


Figure 7-9. *Sample cluster analysis results (Courtesy: Scikit-Learn)*

This cluster analysis depicts three clusters among the data points, which are visualized using the different colors. An important point to remember here is that clustering is an unsupervised learning technique and, from Figure 7-9, it is pretty clear that there will always be some overlap among the clusters since there exists no such definition of a perfect cluster. All the techniques are based on math, heuristics, and some inherent attributes related to generating clusters. They are never a 100% perfect. Hence, there exist several techniques or methods in finding clusters. Some of the more popular clustering algorithms are briefly described as follows:

- **Hierarchical clustering models:** These clustering models are also known as *connectivity-based clustering methods* and they are based on the concept that similar objects will be closer in the vector space and unrelated objects will be farther away. Clusters are formed by connecting objects based on their distance and they can be visualized using a dendrogram. The output of these models is a complete exhaustive hierarchy of clusters. They are subdivided into agglomerative and divisive clustering models.

- **Partition-based or centroid-based clustering models:** These models build clusters in such a way that each cluster has a central representative member that represents each cluster and has the features that distinguish that particular cluster from the rest. There are various algorithms in this, such as k-means, k-mediods, and so on where we need to set the number of clusters (**k**) in advance. Distance metrics, like squares of distances from each data point to the centroid, need to be minimized. The disadvantage of these models is that you need to specify the **k** number of clusters in advance, which may lead to local minima and you might not get a true clustered representation of your data.
- **Distribution-based clustering models:** These models use concepts from probability distributions when clustering data points. The idea is that objects with similar distributions can be clustered into the same group or cluster. Gaussian Mixture Models (GMM) use algorithms like the Expectation-Maximization algorithm for building these clusters. Feature and attribute correlations and dependencies can be captured using these models also, but it is prone to overfitting.
- **Density-based clustering models:** These clustering models generate clusters from data points, which are grouped together at areas of high density compared to the rest of the data points, which may occur randomly across the vector space in sparsely populated areas. These sparse areas are treated as noise and are used as border points to separate clusters. Two popular algorithms in this area include DBSCAN and OPTICS.

There are also several other newer clustering models, like BIRCH and CLARANS. Entire books and journals have been written just on clustering alone, as it is a really interesting topic with a lot of value. Covering every method would be impossible for us in the current scope, hence, we will cover a total of three clustering algorithms and illustrate them with real-world data for better understanding:

- K-means clustering
- Affinity propagation
- Ward's Agglomerative Hierarchical clustering

We cover each algorithm's theoretical concepts as we have done previously with other methods. We also apply each clustering algorithm to real-world data pertaining to movies and their descriptions from the TMDB movie dataset we used in the previous section.

Clustering Movies

We will be clustering a total of 4,800 movies, which we previously cleaned and preprocessed. It's available as the dataframe `df` and the preprocessed corpus is available as the variable `norm_corpus`. The main idea is to cluster these movies into groups using their descriptions as raw input. We will extract features from these description like TF-IDF or document similarity and use unsupervised learning algorithms on them to cluster them. The movie titles we will be showing in the output are just for representation and will be useful when we want to see the movies in each cluster. The data to be fed to the clustering algorithms are the features extracted from the movie descriptions, just to make things clearer.

Feature Engineering

Before we can jump into each of the clustering methods, we will follow the same process of text preprocessing and feature engineering as before. We have already done preprocessing during the document similarity analysis in the previous section, hence we will be using the same `norm_corpus` variable containing our preprocessed movie descriptions. We will now extract bag of words-based features similar to what we did during the document similarity computations, but with some modifications.

```
import nltk
from sklearn.feature_extraction.text import CountVectorizer

stop_words = nltk.corpus.stopwords.words('english')
stop_words = stop_words + ['one', 'two', 'get']

cv = CountVectorizer(ngram_range=(1, 2), min_df=10, max_df=0.8, stop_
words=stop_words)
cv_matrix = cv.fit_transform(norm_corpus)
cv_matrix.shape
```

```
(4800, 3012)
```

Based on the code and output depicted in the preceding snippet, we keep text tokens in our normalized text and extract bag of words count based features for unigrams and bigrams such that each feature occurs in at least 10 documents and at most 80% of the documents using the terms `min_df` and `max_df`. We can see that we have a total of 4,800 rows for the 4,800 movies and a total of 3,012 features for each movie. Now that we have our features and documents ready, we can start the clustering analysis.

K-Means Clustering

The k-means clustering algorithm is a centroid-based clustering model that tries to cluster data into groups or clusters of equal variance. The criteria or measure that this algorithm tries to minimize is *inertia*, also known as *within-cluster sum-of-squares*. Perhaps the one main disadvantage of this algorithm is that the number of clusters (**k**) needs to be specified in advance, as it is with all centroid-based clustering models. This algorithm is perhaps the most popular clustering algorithm, due to its ease of use as well as it being scalable with large amounts of data.

We can now formally define the k-means clustering algorithm along with its mathematical notations. Say we have a dataset **X** with **N** data points or samples and we want to group them into K clusters, where **K** is a user-specified parameter. The k-means clustering algorithm will segregate the **N** data points into **K** disjoint separate clusters, C_k, and each of these clusters can be described by the means of the cluster samples. These means become the cluster centroids μ_k such that these centroids are not bound by the condition that they have to be actual data points from the **N** samples in **X**. The algorithm chooses these centroids and builds the clusters in such a way that the inertia or within-cluster sums of squares are minimized. Mathematically this can be represented as follows:

$$\min \sum_{i=1}^{K} \sum_{x_n \in C_i} \| x_n - \mu_i \|^2$$

with regards to clusters C_i and centroids μ_i such that $i \in \{1, 2, \ldots, k\}$. This optimization is an NP hard problem for all you algorithm enthusiasts out there. Lloyd's algorithm is a solution to this problem. It’s an iterative procedure consisting of the following steps.

1. Choose initial **k** centroids μ_k by taking **k** random samples from the dataset **X**.

2. Update clusters by assigning each data point or sample to its nearest centroid point. Mathematically we can represent this as follows:

   $$C_k = \{x_n : \|x_n - \mu_k\| \leq all \, \|x_n - \mu_l\|\}$$

   where C_k denotes the clusters.

3. Recalculate and update clusters based on the new cluster data points for each cluster obtained from Step 2. Mathematically this can be represented as follows:

   $$\mu_k = \frac{1}{C_k} \sum_{x_n \in C_k} x_n$$

   where μ_k denotes the centroids.

These steps are repeated in an iterative fashion until the outputs of Step 2 and 3 no longer change. One caveat of this method is that even though the optimization is guaranteed to converge, it might lead to a local minimum. Hence, in reality, this algorithm is run multiple times with several epochs and iterations and the results might be averaged from them if needed.

The convergence and occurrence of local minimum are highly dependent on the initialization of the initial centroids in Step 1. One way is to make multiple iterations with multiple random initializations and take the average. Another way would be to use the kmeans++ scheme, as implemented in Scikit-Learn, which initializes the initial centroids to be far apart from each other and has proven to be effective. We now use k-means clustering to cluster the movie data.

```
from sklearn.cluster import KMeans

NUM_CLUSTERS = 6
km = KMeans(n_clusters=NUM_CLUSTERS, max_iter=10000, n_init=50, random_
state=42).fit(cv_matrix)
km

KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=10000,
       n_clusters=6, n_init=50, n_jobs=None, precompute_distances='auto',
       random_state=42, tol=0.0001, verbose=0)

df['kmeans_cluster'] = km.labels_
```

This snippet uses our implemented k-means function to cluster the movies based on the bag of words features from the movie descriptions. We assign the cluster label for each movie from the outcome of this cluster analysis by storing it in the `df` dataframe in the `kmeans_cluster` column. You can see that we have taken **k** to be 6 in our analysis. We can now view the total number of movies for each of the six clusters using the following snippet.

```
# viewing distribution of movies across the clusters
from collections import Counter
Counter(km.labels_)
```

```
Counter({2: 429, 1: 2832, 3: 539, 5: 238, 4: 706, 0: 56})
```

You can see that there are six cluster labels, as expected, 0 to 5, and each has some movies belonging to the cluster. It looks like we have a good distribution of movies in each cluster. We will now do some deeper analysis of our clustering results by showcasing important features that were responsible for the movies being clustered together. We will also look at some of the most popular movies from each cluster!

```
movie_clusters = (df[['title', 'kmeans_cluster', 'popularity']]
                  .sort_values(by=['kmeans_cluster', 'popularity'],
                               ascending=False)
                  .groupby('kmeans_cluster').head(20))
movie_clusters = movie_clusters.copy(deep=True)

feature_names = cv.get_feature_names()
topn_features = 15
ordered_centroids = km.cluster_centers_.argsort()[:, ::-1]

# get key features for each cluster
# get movies belonging to each cluster
for cluster_num in range(NUM_CLUSTERS):
    key_features = [feature_names[index]
                        for index in ordered_centroids[cluster_num, :topn_
                        features]]
    movies = movie_clusters[movie_clusters['kmeans_cluster'] ==
    cluster_num]['title'].values.tolist()
    print('CLUSTER #'+str(cluster_num+1))
    print('Key Features:', key_features)
    print('Popular Movies:', movies)
    print('-'*80)
CLUSTER #1
Key Features: ['film', 'movie', 'story', 'first', 'love', 'making',
'director', 'new', 'time', 'feature', 'made', 'young', '3d', 'american',
'america']
Popular Movies: ['Contact', 'Snatch', 'The Pianist', 'Boyhood', 'Tropic
Thunder', 'Movie 43', 'Night of the Living Dead', 'Almost Famous', 'My
Week with Marilyn', 'Jackass 3D', 'Inside Job', 'Grindhouse', 'The Young
Victoria', 'Disaster Movie', 'Jersey Boys', 'Seed of Chucky', 'Bowling for
Columbine', 'Walking With Dinosaurs', 'Me and You and Everyone We Know',
'Urban Legends: Final Cut']
---------------------------------------------------------------------------
CLUSTER #2
Key Features: ['young', 'man', 'story', 'love', 'family', 'find', 'must',
'time', 'back', 'friends', 'way', 'years', 'help', 'father', 'take']
Popular Movies: ['Interstellar', 'Guardians of the Galaxy', 'Pirates of the
Caribbean: The Curse of the Black Pearl', 'Dawn of the Planet of the Apes',
'The Hunger Games: Mockingjay - Part 1', 'Big Hero 6', 'Whiplash', 'The
Martian', 'Frozen', "Pirates of the Caribbean: Dead Man's Chest", 'Gone
Girl', 'X-Men: Apocalypse', 'Rise of the Planet of the Apes', 'The Lord
of the Rings: The Fellowship of the Ring', 'Pirates of the Caribbean: On
Stranger Tides', "One Flew Over the Cuckoo's Nest", 'Star Wars', 'Brave',
'The Lord of the Rings: The Return of the King', 'Pulp Fiction']
---------------------------------------------------------------------------
CLUSTER #3
Key Features: ['world', 'young', 'find', 'story', 'man', 'new', 'must',
'save', 'way', 'time', 'life', 'evil', 'love', 'family', 'finds']
Popular Movies: ['Minions', 'Jurassic World', 'Captain America: Civil War',
'Avatar', 'The Avengers', "Pirates of the Caribbean: At World's End",
'The Maze Runner', 'Tomorrowland', 'Ant-Man', 'Spirited Away', 'Chappie',
'Monsters, Inc.', 'The Matrix', 'Man of Steel', 'Skyfall', 'The Adventures
of Tintin', 'Nightcrawler', 'Allegiant', 'V for Vendetta', 'Penguins of
Madagascar']
---------------------------------------------------------------------------
```

```
CLUSTER #4
Key Features: ['new', 'york', 'new york', 'city', 'young', 'family',
'love', 'man', 'york city', 'find', 'friends', 'years', 'home', 'must',
'story']
Popular Movies: ['Terminator Genisys', 'Fight Club', 'Teenage Mutant Ninja
Turtles', 'Pixels', 'Despicable Me 2', 'Avengers: Age of Ultron', 'Night at
the Museum: Secret of the Tomb', 'Batman Begins', 'The Dark Knight Rises',
'The Lord of the Rings: The Two Towers', 'The Godfather: Part II', 'How to
Train Your Dragon 2', '12 Years a Slave', 'The Wolf of Wall Street', 'Men
in Black II', "Pan's Labyrinth", 'The Bourne Legacy', 'The Amazing Spider-
Man 2', 'The Devil Wears Prada', 'Non-Stop']
---------------------------------------------------------------------------
CLUSTER #5
Key Features: ['life', 'love', 'man', 'family', 'story', 'young', 'new',
'back', 'years', 'finds', 'hes', 'time', 'find', 'way', 'father']
Popular Movies: ['Deadpool', 'Mad Max: Fury Road', 'Inception', 'The
Godfather', 'Forrest Gump', 'The Shawshank Redemption', 'Harry Potter and
the Chamber of Secrets', 'Inside Out', 'Twilight', 'Maleficent', "Harry
Potter and the Philosopher's Stone", 'Bruce Almighty', 'The Hobbit: An
Unexpected Journey', 'The Twilight Saga: Eclipse', 'Titanic', 'Fifty Shades
of Grey', 'Blade Runner', 'Psycho', 'Up', 'The Lion King']
---------------------------------------------------------------------------
CLUSTER #6
Key Features: ['war', 'world', 'world war', 'ii', 'war ii', 'story',
'young', 'man', 'love', 'army', 'find', 'american', 'battle', 'first',
'must']
Popular Movies: ['The Dark Knight', 'Batman v Superman: Dawn of Justice',
'The Imitation Game', 'Fury', 'The Hunger Games: Mockingjay - Part
2', 'X-Men: Days of Future Past', 'Transformers: Age of Extinction',
"Schindler's List", 'The Good, the Bad and the Ugly', 'American Sniper',
'Thor', 'Shutter Island', 'Underworld', 'Indiana Jones and the Kingdom
of the Crystal Skull', 'Captain America: The First Avenger', 'The Matrix
Revolutions', 'Inglourious Basterds', '300: Rise of an Empire', 'The Matrix
Reloaded', 'Oblivion']
---------------------------------------------------------------------------
```

This output depicts the key features for each cluster and the movies in each cluster. Each cluster is depicted by the main themes, which define that cluster by its top features. You can see popular movies being clustered together based on some key common features. Can you notice any interesting patterns?

We can also use other feature schemes, like pairwise document similarity, to group similar movies in clusters. The following code helps achieve this.

```
from sklearn.metrics.pairwise import cosine_similarity

cosine_sim_features = cosine_similarity(cv_matrix)
km = KMeans(n_clusters=NUM_CLUSTERS, max_iter=10000, n_init=50, random_
state=42).fit(cosine_sim_features)
Counter(km.labels_)

Counter({4: 427, 3: 724, 1: 1913, 2: 504, 0: 879, 5: 353})

df['kmeans_cluster'] = km.labels_

movie_clusters = (df[['title', 'kmeans_cluster', 'popularity']]
                  .sort_values(by=['kmeans_cluster', 'popularity'],
                               ascending=False)
                  .groupby('kmeans_cluster').head(20))
movie_clusters = movie_clusters.copy(deep=True)

# get movies belonging to each cluster
for cluster_num in range(NUM_CLUSTERS):
    movies = movie_clusters[movie_clusters['kmeans_cluster'] == cluster_
    num]['title'].values.tolist()
    print('CLUSTER #'+str(cluster_num+1))
    print('Popular Movies:', movies)
    print('-'*80)
CLUSTER #1
Popular Movies: ['Pirates of the Caribbean: The Curse of the Black Pearl',
'Whiplash', 'The Martian', 'Frozen', 'Gone Girl', 'The Lord of the Rings:
The Fellowship of the Ring', 'Pirates of the Caribbean: On Stranger Tides',
'Pulp Fiction', 'The Fifth Element', 'Quantum of Solace', 'Furious 7',
'Cinderella', 'Man of Steel', 'Gladiator', 'Aladdin', 'The Amazing Spider-
Man', 'Prisoners', 'The Good, the Bad and the Ugly', 'American Sniper',
'Finding Nemo']
---------------------------------------------------------------------------
CLUSTER #2
Popular Movies: ['Interstellar', 'Guardians of the Galaxy', 'Dawn of the
Planet of the Apes', 'The Hunger Games: Mockingjay - Part 1', 'Big Hero 6',
'The Dark Knight', "Pirates of the Caribbean: Dead Man's Chest", 'X-Men:
Apocalypse', 'Rise of the Planet of the Apes', "One Flew Over the Cuckoo's
Nest", 'The Hunger Games: Mockingjay - Part 2', 'Star Wars', 'Brave', 'The
Lord of the Rings: The Return of the King', 'The Hobbit: The Battle of the
Five Armies', 'Iron Man', 'X-Men: Days of Future Past', 'Transformers: Age
of Extinction', 'Spider-Man 3', 'Lucy']
---------------------------------------------------------------------------
CLUSTER #3
Popular Movies: ['Terminator Genisys', 'Fight Club', 'Teenage Mutant Ninja
Turtles', 'Pixels', 'Despicable Me 2', 'Avengers: Age of Ultron', 'Night at
the Museum: Secret of the Tomb', 'Batman Begins', 'The Dark Knight Rises',
'The Lord of the Rings: The Two Towers', 'The Godfather: Part II', 'How to
Train Your Dragon 2', '12 Years a Slave', 'The Wolf of Wall Street', 'Men
in Black II', "Pan's Labyrinth", 'The Bourne Legacy', 'The Amazing Spider-
Man 2', 'The Devil Wears Prada', 'Non-Stop']
---------------------------------------------------------------------------
CLUSTER #4
Popular Movies: ['Deadpool', 'Mad Max: Fury Road', 'Inception', 'The
Godfather', "Pirates of the Caribbean: At World's End", 'Forrest Gump',
'The Shawshank Redemption', 'Harry Potter and the Chamber of Secrets',
'Inside Out', 'Twilight', 'Maleficent', "Harry Potter and the Philosopher's
Stone", 'Bruce Almighty', 'The Hobbit: An Unexpected Journey', 'The
Twilight Saga: Eclipse', 'Fifty Shades of Grey', 'Blade Runner', 'Psycho',
'Up', 'The Lion King']
---------------------------------------------------------------------------
```

```
CLUSTER #5
Popular Movies: ['Minions', 'Jurassic World', 'Captain America: Civil War',
'Batman v Superman: Dawn of Justice', 'Avatar', 'The Avengers', 'Fury',
'The Maze Runner', 'Tomorrowland', 'Ant-Man', 'Spirited Away', 'Chappie',
'Monsters, Inc.', "Schindler's List", 'The Matrix', 'Skyfall', 'The
Adventures of Tintin', 'Nightcrawler', 'Thor', 'Allegiant']
--------------------------------------------------------------------------
CLUSTER #6
Popular Movies: ['The Imitation Game', 'Titanic', 'The Pursuit of
Happyness', 'The Prestige', 'The Grand Budapest Hotel', 'The Fault in Our
Stars', 'Catch Me If You Can', 'Cloud Atlas', 'The Conjuring 2', 'Apollo
13', 'Aliens', 'The Usual Suspects', 'GoodFellas', 'The Princess and the
Frog', 'The Theory of Everything', "The Huntsman: Winter's War", 'Mary
Poppins', 'The Lego Movie', 'Starship Troopers', 'The Big Short']
--------------------------------------------------------------------------
```

Obviously, we used pairwise document similarity as features, hence we do not have specific term-based features that we can depict for each cluster as before. However, we can still see each cluster of similar movies in the preceding output.

Affinity Propagation

The k-means algorithm, while very popular, has the drawback of the user having to define the number of clusters. What if the number of clusters changes? There are some ways of checking the cluster quality and determining the value of the optimum **k** might be. Interested readers can check out the Elbow method and the Silhouette coefficient, which are popular methods of determining the optimum **k**.

Here, we talk about an algorithm that tried to build clusters based on inherent properties of the data without any assumptions about the number of clusters. The Affinity Propagation (AP) algorithm is based on the concept of "message passing" among the various data points to be clustered and there is no assumption about the number of possible clusters.

AP creates these clusters from the data points by passing messages between pairs of data points until convergence is achieved. The entire dataset is then represented by a small number of exemplars, which act as representatives for samples.

These exemplars are analogous to the centroids that you obtain from k-means or k-medoids. The messages sent between pairs represent how suitable one of the points might be, in being the exemplar or representative of the other data point. This keeps getting updated in every iteration until convergence is achieved with the final exemplars being the representatives of each cluster. Remember one drawback of this method is that it is computationally intensive. Messages are passed between each pair of data points across the entire dataset and can take substantial time to converge for large datasets.

We can now define the steps involved in the AP algorithm (courtesy of Scikit-Learn). Consider that we have a dataset **X** with **n** data points, such that $X = \{x_1, x_2, \ldots, x_n\}$, and we let $sim(x, y)$ be the similarity function, which quantifies the similarity between two points **x** and **y**. In our implementation, we use the Cosine similarity again for this. The AP algorithm iteratively proceeds by executing two message-passing steps as follows.

1. Responsibility updates are sent around and can be mathematically represented as

   $$r(i,k) \leftarrow sim(i,k) - \max_{k' \neq k}\{a(i,k') + sim(i,k')\}$$

   where the responsibility matrix is **R** and $r(i, k)$ is a measure that quantifies how well x_k can serve as being the representative or exemplar for x_i in comparison to the other candidates.

2. Availability updates are then sent around that can be mathematically represented as

   $$a(i,k) \leftarrow \min\left(0, r(k,k) + \sum_{i' \notin \{i,k\}} \max(0, r(i',k))\right)$$

   for $i \neq k$ and availability for $i = k$ is represented as

   $$a(k,k) \leftarrow \sum_{i' \neq k} \max(0, r(i',k))$$

   where the availability matrix is **A** and $a(i, k)$ represents how appropriate it would be for x_i to pick x_k as its exemplar considering all the other points preference to pick x_k as an exemplar.

These two steps keep occurring for each iteration until convergence is achieved. One of the main disadvantages of this algorithm is the fact that you might end up with too many clusters. We will showcase only the top ten largest clusters here:

```
from sklearn.cluster import AffinityPropagation

ap = AffinityPropagation(max_iter=1000)
ap.fit(cosine_sim_features)
res = Counter(ap.labels_)
res.most_common(10)

[(183, 1355), (182, 93), (159, 80), (54, 74), (81, 57),
 (16, 51), (26, 47), (24, 45), (48, 43), (89, 42)]
```

Let's now try to showcase the top popular movies for each of the ten clusters (we do not consider the clusters with a smaller number of movies here).

```
df['affprop_cluster'] = ap.labels_
filtered_clusters = [item[0] for item in res.most_common(8)]
filtered_df = df[df['affprop_cluster'].isin(filtered_clusters)]
movie_clusters = (filtered_df[['title', 'affprop_cluster', 'popularity']]
                  .sort_values(by=['affprop_cluster', 'popularity'],
                               ascending=False)
                  .groupby('affprop_cluster').head(20))
movie_clusters = movie_clusters.copy(deep=True)

# get key features for each cluster
# get movies belonging to each cluster
for cluster_num in range(len(filtered_clusters)):
    movies = movie_clusters[movie_clusters['affprop_cluster'] == filtered_
    clusters[cluster_num]]['title'].values.tolist()
    print('CLUSTER #'+str(filtered_clusters[cluster_num]))
    print('Popular Movies:', movies)
    print('-'*80)

CLUSTER #183
Popular Movies: ['Interstellar', 'Dawn of the Planet of the Apes', 'Big
Hero 6', 'The Dark Knight', "Pirates of the Caribbean: Dead Man's Chest",
'The Hunger Games: Mockingjay - Part 2', 'Star Wars', 'Brave', 'The Lord
of the Rings: The Return of the King', 'The Hobbit: The Battle of the Five
Armies', 'Iron Man', 'Transformers: Age of Extinction', 'Lucy', 'Mission:
Impossible - Rogue Nation', 'Maze Runner: The Scorch Trials', 'Spectre',
'The Green Mile', 'Terminator 2: Judgment Day', 'Exodus: Gods and Kings',
'Harry Potter and the Goblet of Fire']
----------------------------------------------------------------------------
CLUSTER #182
Popular Movies: ['Inception', 'Harry Potter and the Chamber of Secrets',
'The Hobbit: An Unexpected Journey', 'Django Unchained', 'American Beauty',
'Snowpiercer', 'Trainspotting', 'First Blood', 'The Bourne Supremacy', 'Yes
Man', 'The Secret Life of Walter Mitty', 'RED', 'Casino', 'The Passion of
the Christ', 'Annie', 'Fantasia', 'Vicky Cristina Barcelona', 'The Butler',
'The Secret Life of Pets', 'Edge of Darkness']
----------------------------------------------------------------------------
CLUSTER #159
Popular Movies: ['Gone Girl', 'Pulp Fiction', 'Gladiator', 'Saving Private
Ryan', 'The Game', 'Jack Reacher', 'The Fugitive', 'The Purge: Election
Year', 'The Thing', 'The Rock', '3:10 to Yuma', 'Wild Card', 'Blackhat',
'Knight and Day', 'Equilibrium', 'Black Hawk Down', 'Immortals', '1408',
'The Call', 'Up in the Air']
----------------------------------------------------------------------------
CLUSTER #54
Popular Movies: ['Despicable Me 2', 'The Lord of the Rings: The Two
Towers', 'The Bourne Legacy', 'Horrible Bosses 2', 'Sherlock Holmes: A
Game of Shadows', "Ocean's Twelve", 'Raiders of the Lost Ark', 'Star Trek
Beyond', 'Fantastic 4: Rise of the Silver Surfer', 'Sherlock Holmes', 'Dead
Poets Society', 'Batman & Robin', 'Madagascar: Escape 2 Africa', 'Paul
Blart: Mall Cop 2', 'Kick-Ass 2', 'Anchorman 2: The Legend Continues', 'The
Pacifier', "The Devil's Advocate", 'Tremors', 'Wild Hogs']
----------------------------------------------------------------------------
CLUSTER #81
Popular Movies: ['Whiplash', 'Sicario', 'Jack Ryan: Shadow Recruit', 'The
Untouchables', 'Young Frankenstein', 'Point Break', '8 Mile', 'The Final
Destination', 'Savages', 'Scooby-Doo', 'The Artist', 'The Last King of
Scotland', 'Sinister 2', 'Another Earth', 'The Darkest Hour', 'Wall Street:
Money Never Sleeps', 'The Score', 'Doubt', 'Revolutionary Road', 'Crimson Tide']
----------------------------------------------------------------------------
```

```
CLUSTER #16
Popular Movies: ['The Shawshank Redemption', 'Inside Out', 'Batman Begins',
'Psycho', 'Cars', 'Ice Age: Dawn of the Dinosaurs', 'The Chronicles of
Narnia: Prince Caspian', 'Kung Fu Panda 2', 'The Witch', 'Madagascar',
'Wild', 'Shame', 'Scream 2', '16 Blocks', 'Last Action Hero', 'Garden
State', '25th Hour', 'The House Bunny', 'The Jacket', 'Any Given Sunday']
---------------------------------------------------------------------------
CLUSTER #26
Popular Movies: ['Minions', 'Avatar', 'Penguins of Madagascar', 'Iron
Man 3', 'London Has Fallen', 'The Great Gatsby', 'Transcendence', 'The
5th Wave', 'Zombieland', 'Hotel Transylvania', 'Ghost Rider: Spirit of
Vengeance', 'Warm Bodies', 'Paul', 'The Road', 'Alexander', 'This Is the
End', "Bridget Jones's Diary", 'G.I. Joe: The Rise of Cobra', 'Hairspray',
'Step Up Revolution']
---------------------------------------------------------------------------
CLUSTER #24
Popular Movies: ['Spider-Man', 'Chronicle', '21 Jump Street', '22 Jump
Street', 'Project X', 'Kick-Ass', 'Grown Ups', 'American Wedding', 'Kiss
Kiss Bang Bang', 'I Know What You Did Last Summer', 'Here Comes the Boom',
'Dazed and Confused', 'Not Another Teen Movie', 'WarGames', 'Fast Times at
Ridgemont High', 'American Graffiti', 'The Gallows', 'Dumb and Dumberer:
When Harry Met Lloyd', 'Bring It On', 'The New Guy']
---------------------------------------------------------------------------
```

An important point to note here is that a few keywords from the exemplars or centroids for each cluster may not always depict the true theme of that cluster. A good idea is to build topic models on each cluster and see what kind of topics you can extract from each cluster that would make a better representation of each cluster (another example where you can see how to connect various text analytics techniques).

Ward's Agglomerative Hierarchical Clustering

The Hierarchical clustering family of algorithms is a bit different from the other clustering models discussed earlier. Hierarchical clustering tries to build a nested hierarchy of clusters by merging or splitting them in succession. There are two main strategies for Hierarchical clustering:

- **Agglomerative:** These algorithms follow a bottom-up approach. All data points initially belong to their own individual cluster and then from this bottom layer, we start merging clusters, thereby building a hierarchy of clusters as we go up.
- **Divisive:** These algorithms follow a top-down approach. All the data points initially belong to a single huge cluster and then we start recursively dividing them up as we gradually move down. This produces a hierarchy of clusters going from the top down.

Merges and splits usually happen using a greedy algorithm and the end result of the hierarchy of clusters can be visualized as a tree structure, called a *dendrogram*. Figure 7-10 shows how a dendrogram is constructed using Agglomerative Hierarchical clustering.

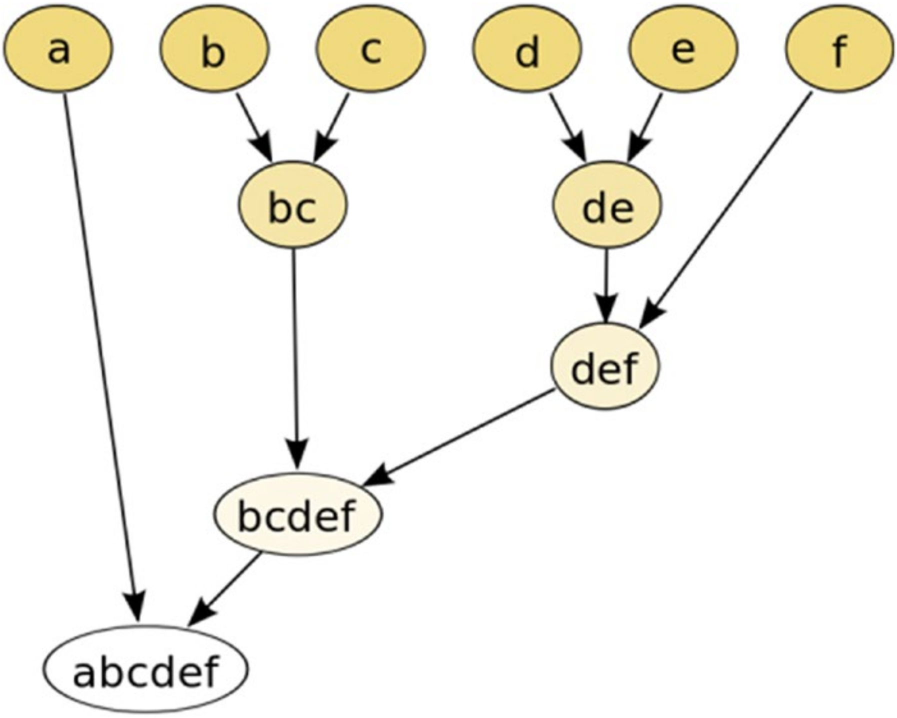


Figure 7-10. *Agglomerative Hierarchical clustering representation*

Figure 7-10 clearly highlights how six separate data points start off as six clusters and then we slowly start grouping them in each step, following a bottom-up approach. We use an Agglomerative Hierarchical clustering algorithm in this section. In the

Agglomerative clustering, for deciding which clusters we should combine when starting from the individual data point clusters, we need two things:

- A *distance metric* to measure the similarity or dissimilarity degree between data points. We will be using the cosine distance/similarity in our implementation
- A *linkage criterion,* which determines the metric to be used for the merging strategy of clusters. We will be using Ward's method here.

The Ward's linkage criterion minimizes the sum of squared differences within all the clusters and is a variance minimizing approach. This is also known as Ward's minimum variance method and was initially presented by J. Ward. The idea is to minimize the variances within each cluster using an objective function like the L2 norm distance between two points. We can start by computing the initial cluster distances between each pair of points using this formula:

$$d_{ij} = d\left(\left\{C_i, C_j\right\}\right) = \| C_i - C_j \|^2$$

where initially C_i indicates cluster **i** with one document and at each iteration. We find the pairs of clusters that lead to the least increase in variance for that cluster once merged. A weighted squared Euclidean distance, or L2 norm, as depicted in the previous formula, would suffice for this algorithm. We use Cosine similarity to compute the cosine distances between each pair of movies for our dataset. The following function implements Ward's Agglomerative Hierarchical clustering:

```
from scipy.cluster.hierarchy import ward, dendrogram
from sklearn.metrics.pairwise import cosine_similarity

def ward_hierarchical_clustering(feature_matrix):

    cosine_distance = 1 - cosine_similarity(feature_matrix)
    linkage_matrix = ward(cosine_distance)
    return linkage_matrix
```

To view the results of the Hierarchical clustering, we need to plot a dendrogram using the linkage matrix. Hence, we implement the following function to build and plot a dendrogram from the Hierarchical clustering linkage matrix.

```
def plot_hierarchical_clusters(linkage_matrix, movie_data, p=100,
figure_size=(8,12)):
    # set size
    fig, ax = plt.subplots(figsize=figure_size)
    movie_titles = movie_data['title'].values.tolist()
    # plot dendrogram
    R = dendrogram(linkage_matrix, orientation="left", labels=movie_titles,
                    truncate_mode='lastp',
                    p=p,
                    no_plot=True)
    temp = {R["leaves"][ii]: movie_titles[ii] for ii in range(len(R["leaves"]))}
    def llf(xx):
        return "{}".format(temp[xx])
    ax = dendrogram(
            linkage_matrix,
            truncate_mode='lastp',
            orientation="left",
            p=p,
            leaf_label_func=llf,
            leaf_font_size=10.,
            )
    plt.tick_params(axis= 'x',
                    which='both',
                    bottom='off',
                    top='off',
                    labelbottom='off')
    plt.tight_layout()
    plt.savefig('movie_hierachical_clusters.png', dpi=200)
```

We are now ready to perform hierarchical clustering on our movie data! The following code snippet shows Ward's clustering in action.

```
linkage_matrix = ward_hierarchical_clustering(cv_matrix)
plot_hierarchical_clusters(linkage_matrix,
                           p=100,
                           movie_data=df,
                           figure_size=(12, 14))
```

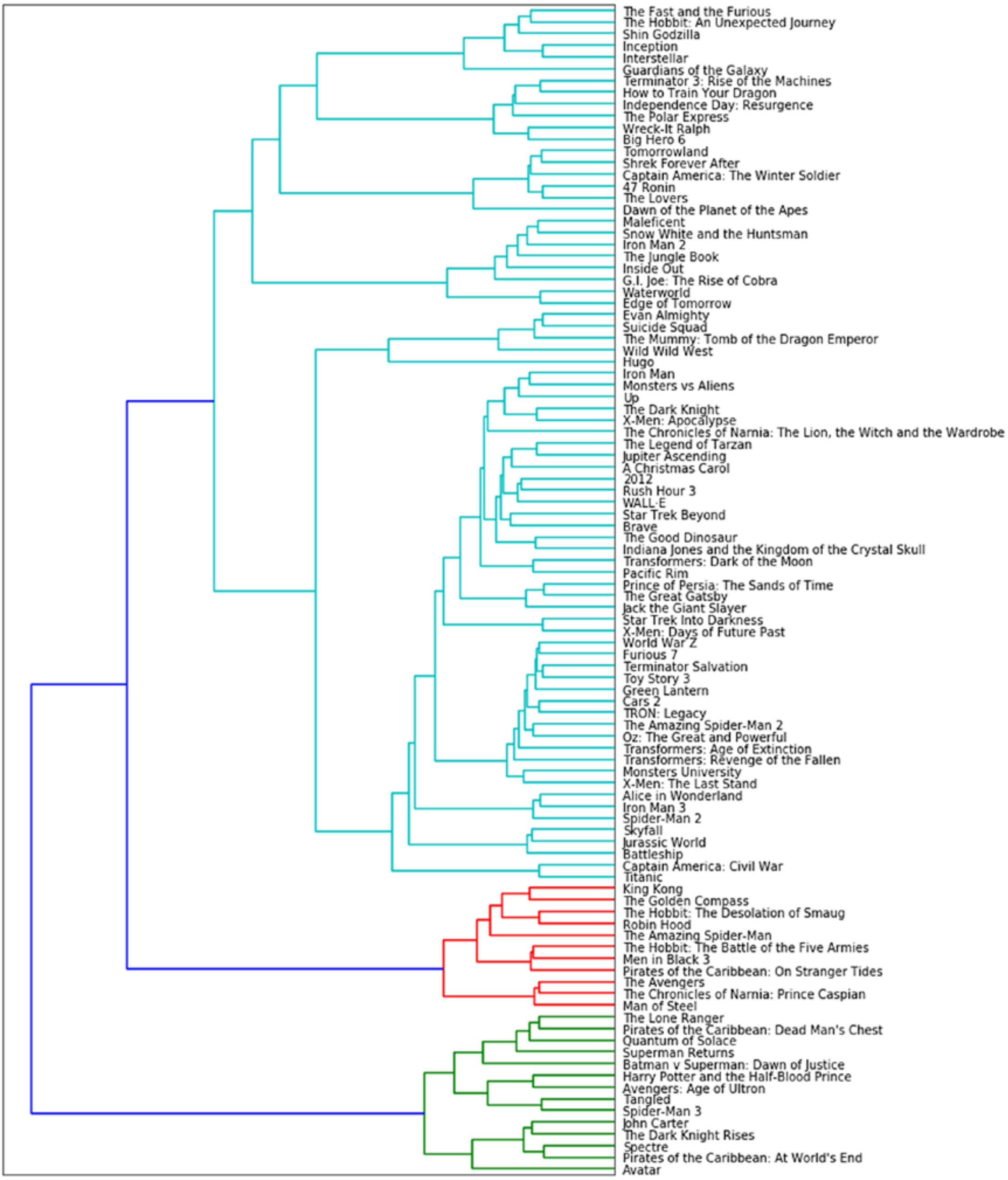


Figure 7-11. *Ward's clustering dendrogram on our movies*

The dendrogram in Figure 7-11 shows us the clustering analysis results. The colors indicate there are three main clusters, which are further subdivided into more granular clusters, thereby maintaining a hierarchy. If you have trouble reading the small fonts or can't see the colors, you can view the same figure in the file named `movie_hierachical_clusters.png` available with the code files in this chapter. We only show the last 100 movies in the dendrogram due to the lack of space for visualization.

Summary

We covered a lot of content in this chapter, including several topics in the challenging but very interesting unsupervised machine learning domain. You now know how text similarity can be computed and various kinds of distance measures and metrics. We also looked at important concepts related to distance metrics and measures and properties that make a measure into a metric. We also looked at concepts related to unsupervised machine learning and how we can incorporate such techniques in document clustering.

Various ways of measuring term and document similarity were also covered and we also implemented several of these techniques by successfully converting mathematical equations into code using the power of Python and several open source libraries. We touched upon document clustering in detail, looking at the various concepts and types of clustering models.

Finally we took a real-world example of clustering the top-100 greatest movies of all time using IMDB movie synopses data and used different clustering models like k-means, affinity propagation, and Ward's hierarchical clustering to build, analyze, and visualize clusters. This should be enough for you to get started analyzing document similarity and clustering. You can even start combining various techniques from the chapters covered so far. (Hint: Topic models with clustering, building classifiers by combining supervised and unsupervised learning, and augmenting recommendation systems using document clusters to just name a few!)

CHAPTER 8

Semantic Analysis

Natural language understanding has gained significant importance in the last decade with the advent of machine learning and further advances like deep learning and artificial intelligence. Computers, or machines in general, can be programmed to learn specific things or perform specific operations. However, the key limitation is their inability to perceive, understand, and comprehend things like humans do. With the resurgence in popularity of neural networks and advances made in computer architecture, we now have deep learning and artificial intelligence evolving at a rapid pace and we have been engineering machines into learning, perceiving, understanding, and performing actions on their own. You may have seen or heard several of these efforts in the form of self-driving cars, computers beating experienced players in their own games like Chess and Go, and more recently chatbots. So far, we have looked at various computational, language processing, and machine learning techniques to classify, cluster, and summarize text. We also developed certain methods and programs to analyze and understand text syntax and structure. This chapter deals with methods that try to answer the question, "Can we analyze and understand the meaning and sentiment behind a body of text?"

Natural language processing has a wide variety of applications. People try to use natural language understanding to infer the meaning and context behind text and to solve various problems. We discussed several of these applications briefly in Chapter 1. To refresh your memory, the following applications require extensive understanding from the semantic perspective.

- Question answering systems
- Contextual recognition
- Speech recognition (for some applications)

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_8

Text semantics specifically deals with understanding the meaning of text or language. Words, when combined into sentences, have some lexical relations and contextual relations. This leads to various types of relationships and hierarchies. Semantics sits at the heart of all this in that it tries to analyze and understand these relationships and infer meaning from them. We explore various types of semantic relationships in natural language and look at some NLP-based techniques for inferring and extracting meaningful semantic information from text. Semantics is purely concerned with context and meaning and the textual structure holds little significance here. However, sometimes the syntax or arrangement of words helps us infer the context of words and helps us differentiate things like "lead" as a metal from "lead" as in the lead of a movie!

In this chapter, we cover several aspects of semantic analysis. We start by exploring WordNet, which is a lexical database, and introduce a new concept called *synsets*. We also explore various semantic relationships and representations in natural language. We cover techniques like word sense disambiguation and named entity recognition (NER), wherein you will even build your own NER from scratch! Let's get started. All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Semantic Analysis

We have seen how terms or words are grouped into phrases, which further form clauses and finally sentences. Chapter 3 showed us various structural components in natural language, including parts of speech, chunking, and grammar. All these concepts fall under the syntactic and structural analysis of text data. While we do explore relationships of words, phrases, and clauses, these are purely based on their position, syntax, and structure. Semantic analysis is more about understanding the actual context and meaning behind words in text and how they relate to other words and convey some information as a whole.

As mentioned in Chapter 1, *semantics* is the study of meaning. *Linguistic semantics* is a complete branch under linguistics that deals with studying of meaning in natural language. This includes exploring relationships between words, phrases, and symbols.

Besides this, there are also various ways to represent semantics associated with statements and propositions. We broadly cover the following topics under semantic analysis:

- Exploring WordNet and synsets
- Analyzing lexical semantic relations
- Word sense disambiguation
- Named entity recognition
- Analyzing semantic representations

The main objective of these topics is to give you a clear understanding of the methodologies that you can leverage for semantic analysis as well as understand how to use them. You can refresh your memory by revisiting the "Language Semantics" section in Chapter 1. We will be revisiting several concepts from there again in this chapter with real-world examples. Without any further delay, let's get started!

Exploring WordNet

WordNet is a huge lexical database for the English language. This database is a part of Princeton University and you can get more detailed information at `https://wordnet.princeton.edu/`, which is the official website for WordNet. It was originally created in 1985, in the Princeton University's Cognitive Science Laboratory under the direction of Professor G.A. Miller. This lexical database consists of nouns, adjectives, verbs, and adverbs. Related lexical terms are grouped into sets based on common concepts. These sets are known as cognitive synonym sets or *synsets* and each expresses a unique, distinct concept.

At a high level, WordNet can be compared to a thesaurus or a dictionary that provides words and their synonyms, but on a lower level, it is much more than that. Synsets and their corresponding terms have detailed relationships and hierarchies based on their semantic meaning. WordNet is used extensively as a lexical database, in text analytics, natural language processing, and artificial intelligence based applications.

The WordNet database consists of over 155,000 words and they are represented in across 117,000 synsets and contain over 206,000 word-sense pairs. The database is roughly 12MB and can be accessed through various interfaces and APIs. The official website has a web application interface for accessing various details related to words, synsets, and concepts. You can access it at http://wordnetweb.princeton.edu/perl/webwn or download it from https://wordnet.princeton.edu/wordnet/download/, which contained various packages, files, and tools related to WordNet. We will be accessing WordNet programmatically using the interface provided by the NLTK package. We start by exploring synsets and then various semantic relationships using synsets.

Understanding Synsets

We start exploring WordNet by looking at synsets, since they are perhaps one of the most important structures and they tie everything together. In general, based on concepts from NLP and information retrieval, synsets are defined as a collection of data entities that are considered semantically similar. This doesn't mean that they will be exactly the same, but they will be centered on similar context and concepts. Specifically in the context of WordNet, synsets are defined as a set or collection of synonyms that are interchangeable and revolve around a specific concept. Synsets not only consist of simple words but also collocations. Polysemous word forms (words that sound and look the same but have different but relatable meanings) are assigned to different synsets based on their meaning. Synsets are connected to other synsets using semantic relations, which we explore in a future section. Typically each synset has the term, a definition explaining the meaning of the term, some optional examples, and related lemmas (collection of synonyms) to the term. Some terms may have multiple synsets associated with them, where each synset has a particular context.

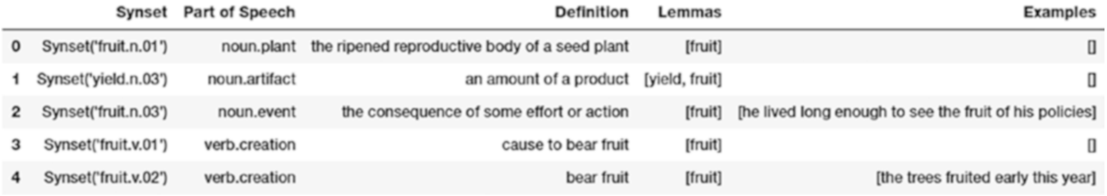
Let's look at a real example by using NLTK's WordNet interface to explore synsets associated with the word, "fruit". We can do this using the following code snippet.

```
from nltk.corpus import wordnet as wn
import pandas as pd

term = 'fruit'
synsets = wn.synsets(term)
# display total synsets
print 'Total Synsets:', len(synsets)
```

```
Total Synsets: 5
```

We can see that there are a total of five synsets associated with the word “fruit”. What can these synsets indicate? We can dig deeper into each synset and its components using the following code snippet (see Figure 8-1).

```
pd.options.display.max_colwidth = 200
fruit_df = pd.DataFrame([{'Synset': synset,
                          'Part of Speech': synset.lexname(),
                          'Definition': synset.definition(),
                          'Lemmas': synset.lemma_names(),
                          'Examples': synset.examples()}
                             for synset in synsets])
fruit_df = fruit_df[['Synset', 'Part of Speech', 'Definition', 'Lemmas',
'Examples']]
fruit_df
```



	Synset	Part of Speech	Definition	Lemmas	Examples
0	Synset('fruit.n.01')	noun.plant	the ripened reproductive body of a seed plant	[fruit]	[]
1	Synset('yield.n.03')	noun.artifact	an amount of a product	[yield, fruit]	[]
2	Synset('fruit.n.03')	noun.event	the consequence of some effort or action	[fruit]	[he lived long enough to see the fruit of his policies]
3	Synset('fruit.v.01')	verb.creation	cause to bear fruit	[fruit]	[]
4	Synset('fruit.v.02')	verb.creation	bear fruit	[fruit]	[the trees fruited early this year]

Figure 8-1. *Exploring WordNet synsets*

The output shows us details pertaining to each synset associated with the word “fruit” and the definitions give us the sense of each synset and the lemma associated with it. The part of speech for each synset is also mentioned, which includes nouns and verbs. Some examples in the output depict how the term is used in actual sentences. Now that we understand synsets better, let’s start exploring various semantic relationships.

Analyzing Lexical Semantic Relationships

Text semantics itself indicates the study of meaning and context. Synsets give a nice abstraction over various terms and provide useful information like definition, examples, parts of speech, and lemmas. But can we explore semantic relationships among entities using synsets? The answer is definitely yes. We cover many of the concepts related to semantic relations that we covered in detail in the "Lexical Semantic Relations" subsection under the "Language Semantics" section in Chapter 1. It would be useful for you to skim through that section to understand each of the concepts better when we illustrate them with real-world examples here. We use NLTK's WordNet resource here, but you can also use the same WordNet resource from the `pattern` package, as it includes a similar interface to NLTK.

Entailments

The term *entailment* usually refers to some event or action that logically involves or is associated with some other action or event that has taken place or will take place. Ideally this applies very well to verbs indicating some specific action. The following snippet shows how to get entailments.

```
# entailments
for action in ['walk', 'eat', 'digest']:
    action_syn = wn.synsets(action, pos='v')[0]
    print(action_syn, '-- entails -->', action_syn.entailments())
```

```
Synset('walk.v.01') -- entails --> [Synset('step.v.01')]
Synset('eat.v.01') -- entails --> [Synset('chew.v.01'),
Synset('swallow.v.01')]
Synset('digest.v.01') -- entails --> [Synset('consume.v.02')]
```

You can see how related synsets depict the concept of entailment in this output. Related actions are depicted in entailment, where actions like walking involve or entail stepping and eating entails chewing and swallowing.

Homonyms and Homographs

On a high level, *homonyms* refer to words having the same written form or pronunciation but with different meanings. They are a superset of *homographs*, which are words with the same spelling but with different pronunciation or meanings. The following code snippet shows us how we can get homonyms/homographs.

```
for synset in wn.synsets('bank'):
    print(synset.name(),'-',synset.definition())
```

```
bank.n.01 - sloping land (especially the slope beside a body of water)
depository_financial_institution.n.01 - a financial institution that
accepts deposits and channels the money into lending activities
bank.n.03 - a long ridge or pile
bank.n.04 - an arrangement of similar objects in a row or in tiers
...
...
deposit.v.02 - put into a bank account
bank.v.07 - cover with ashes so to control the rate of burning
trust.v.01 - have confidence or faith in
```

This output shows a part of the result obtained for the various homographs for the word "bank". You can see that there are various different meanings associated with the word "bank," which is the core idea behind homographs.

Synonyms and Antonyms

Synonyms are words with similar meanings and *antonyms* are words with opposite or contrasting meanings, as you might know already. The following snippet depicts synonyms and antonyms.

```
term = 'large'
synsets = wn.synsets(term)
adj_large = synsets[1]
adj_large = adj_large.lemmas()[0]
adj_large_synonym = adj_large.synset()
adj_large_antonym = adj_large.antonyms()[0].synset()

print('Synonym:', adj_large_synonym.name())
print('Definition:', adj_large_synonym.definition())
print('Antonym:', adj_large_antonym.name())
print('Definition:', adj_large_antonym.definition())
print()
```

```
Synonym: large.a.01
Definition: above average in size or number or quantity or magnitude or extent
Antonym: small.a.01
Definition: limited or below average in number or quantity or magnitude or extent
```

```
term = 'rich'
synsets = wn.synsets(term)[:3]

for synset in synsets:
    rich = synset.lemmas()[0]
    rich_synonym = rich.synset()
    rich_antonym = rich.antonyms()[0].synset()

    print('Synonym:', rich_synonym.name())
    print('Definition:', rich_synonym.definition())
    print('Antonym:', rich_antonym.name())
    print('Definition:', rich_antonym.definition())
    print()
```

```
Synonym: rich_people.n.01
Definition: people who have possessions and wealth (considered as a group)
Antonym: poor_people.n.01
Definition: people without possessions or wealth (considered as a group)

Synonym: rich.a.01
Definition: possessing material wealth
Antonym: poor.a.02
Definition: having little money or few possessions

Synonym: rich.a.02
Definition: having an abundant supply of desirable qualities or substances
(especially natural resources)
Antonym: poor.a.04
Definition: lacking in specific resources, qualities or substances
```

These outputs show sample synonyms and antonyms for the word “large” and the word “rich”. Additionally, we explore several synsets associated with the term or concept “rich,” which give us distinct synonyms and their corresponding antonyms.

Hyponyms and Hypernyms

Synsets represent terms with unique semantics and concepts and they are related to each other based on some similarity. Several of these synsets also represent abstract and generic concepts, besides concrete entities. Usually they are interlinked in the form of a hierarchical structure representing “is-a” relationships. *Hyponyms* and *hypernyms* help us explore related concepts by navigating through this hierarchy. To be more specific, hyponyms refer to entities or concepts that are a subclass of a higher order concept and have very specific sense or context compared to their superclass. The following snippet shows the hyponyms for the word “tree”.

```
term = 'tree'
synsets = wn.synsets(term)
tree = synsets[0]

print('Name:', tree.name())
print('Definition:', tree.definition())
```

```
Name: tree.n.01
Definition: a tall perennial woody plant having a main trunk and branches
forming a distinct elevated crown; includes both gymnosperms and
angiosperms
```

```
hyponyms = tree.hyponyms()
print('Total Hyponyms:', len(hyponyms))
print('Sample Hyponyms')
for hyponym in hyponyms[:10]:
    print(hyponym.name(), '-', hyponym.definition())
    print()
```

```
Total Hyponyms: 180
Sample Hyponyms
aalii.n.01 - a small Hawaiian tree with hard dark wood

acacia.n.01 - any of various spiny trees or shrubs of the genus Acacia

african_walnut.n.01 - tropical African timber tree with wood that resembles
mahogany

albizzia.n.01 - any of numerous trees of the genus Albizia

alder.n.02 - north temperate shrubs or trees having toothed leaves and
conelike fruit; bark is used in tanning and dyeing and the wood is rot-
resistant
```

```
angelim.n.01 - any of several tropical American trees of the genus Andira

angiospermous_tree.n.01 - any tree having seeds and ovules contained in the
ovary

anise_tree.n.01 - any of several evergreen shrubs and small trees of the
genus Illicium

arbor.n.01 - tree (as opposed to shrub)

aroeira_blanca.n.01 - small resinous tree or shrub of Brazil
```

This output tells us that there are a total of 180 hyponyms for the word “tree” and we see some of the sample hyponyms and their definitions. We can see that each hyponym is a specific type of tree, as expected. Hyponyms are entities or concepts that act as superclasses to hyponyms and have a more generic sense or context. The following snippet shows the immediate superclass hyponym for “tree”.

```
hypernyms = tree.hypernyms()
print(hypernyms)
```

```
[Synset('woody_plant.n.01')]
```

You can even navigate up the entire entity/concept hierarchy and depict all the hyponyms or parent classes for “tree” by using the following code snippet.

```
# get total hierarchy pathways for 'tree'
hypernym_paths = tree.hypernym_paths()
print('Total Hypernym paths:', len(hypernym_paths))
```

```
Total Hypernym paths: 1
```

```
# print the entire hypernym hierarchy
print('Hypernym Hierarchy')
print(' -> '.join(synset.name() for synset in hypernym_paths[0]))
```

```
Hypernym Hierarchy
entity.n.01 -> physical_entity.n.01 -> object.n.01 -> whole.n.02 -> living_
thing.n.01 -> organism.n.01 -> plant.n.02 -> vascular_plant.n.01 -> woody_
plant.n.01 -> tree.n.01
```

From this output, you can see that entity is the most generic concept in which “tree” is present and the complete hypernym hierarchy showing the corresponding hypernym or superclass at each level is shown. As you navigate further down, you get more specific concepts/entities, and if you go in the reverse direction, you will get more generic concepts/entities.

Holonyms and Meronyms

Holonyms contain a specific entity of interest. Basically, they are defined as the relationship between entity that denotes the whole and a term denoting a specific part of the whole. The following snippet shows holonyms for “tree”.

```
member_holonyms = tree.member_holonyms()
print('Total Member Holonyms:', len(member_holonyms))
print('Member Holonyms for [tree]:-')
for holonym in member_holonyms:
    print(holonym.name(), '-', holonym.definition())
    print()
```

```
Total Member Holonyms: 1
Member Holonyms for [tree]:-
forest.n.01 - the trees and other plants in a large densely wooded area
```

From the output, we can see that “forest” is a holonym for “tree,” which is semantically correct. This makes sense because a forest is a collection of trees. *Meronyms* are semantic relationships that relate a term or entity as a part or constituent of another term or entity. The following snippet depicts different types of meronyms for the word tree.

```
part_meronyms = tree.part_meronyms()
print('Total Part Meronyms:', len(part_meronyms))
print('Part Meronyms for [tree]:-')
for meronym in part_meronyms:
    print(meronym.name(), '-', meronym.definition())
    print()
```

```
Total Part Meronyms: 5
Part Meronyms for [tree]:-
burl.n.02 - a large rounded outgrowth on the trunk or branch of a tree
crown.n.07 - the upper branches and leaves of a tree or other plant
```

```
limb.n.02 - any of the main branches arising from the trunk or a bough of a tree
stump.n.01 - the base part of a tree that remains standing after the tree
has been felled
trunk.n.01 - the main stem of a tree; usually covered with bark; the bole
is usually the part that is commercially useful for lumber
```

```
# substance based meronyms for tree
substance_meronyms = tree.substance_meronyms()
print('Total Substance Meronyms:', len(substance_meronyms))
print('Substance Meronyms for [tree]:-')
for meronym in substance_meronyms:
    print(meronym.name(), '-', meronym.definition())
    print()
```

```
Total Substance Meronyms: 2
Substance Meronyms for [tree]:-
heartwood.n.01 - the older inactive central wood of a tree or woody plant;
usually darker and denser than the surrounding sapwood
sapwood.n.01 - newly formed outer wood lying between the cambium and the
heartwood of a tree or woody plant; usually light colored; active in water
conduction
```

This output depicts meronyms that include various constituents of trees, like “stump” and “trunk” and various derived substances from trees, like “heartwood” and “sapwood”.

Semantic Relationships and Similarity

In the previous sections, we looked at various concepts related to lexical semantic relationships. We will now look at ways to connect similar entities based on their semantic relationships and measure semantic similarity between them. Semantic similarity is different from the conventional similarity metrics we discussed in Chapter 6. We will use some sample synsets related to living entities, as depicted in the following snippet, for our analysis.

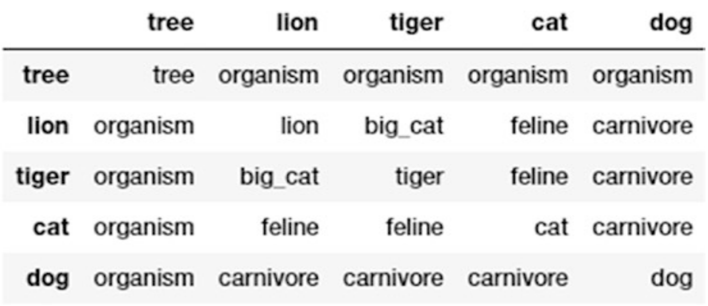
```
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
```

```
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')

# create entities and extract names and definitions
entities = [tree, lion, tiger, cat, dog]
entity_names = [entity.name().split('.')[0] for entity in entities]
entity_definitions = [entity.definition() for entity in entities]

# print entities and their definitions
for entity, definition in zip(entity_names, entity_definitions):
    print(entity, '-', definition)
    print()
```

```
tree - a tall perennial woody plant having a main trunk and branches
forming a distinct elevated crown; includes both gymnosperms and
angiosperms

lion - large gregarious predatory feline of Africa and India having a tawny
coat with a shaggy mane in the male

tiger - large feline of forests in most of Asia having a tawny coat with
black stripes; endangered

cat - feline mammal usually having thick soft fur and no ability to roar:
domestic cats; wildcats

dog - a member of the genus Canis (probably descended from the common wolf)
that has been domesticated by man since prehistoric times; occurs in many
breeds
```

Now that we know our entities a bit better, we will try to correlate these entities based on common hypernyms. For each pair of entities, we will try to find the lowest common hypernym in the relationship hierarchy tree. Correlated entities are expected to have very specific hypernyms and unrelated entities should have very abstract or generic hypernyms. The following code snippet depicts this.

```
common_hypernyms = []
for entity in entities:
    # get pairwise lowest common hypernyms
    common_hypernyms.append([entity.lowest_common_hypernyms(compared_entity)[0]
                                          .name().split('.')[0]
                           for compared_entity in entities])

# build pairwise lower common hypernym matrix
common_hypernym_frame = pd.DataFrame(common_hypernyms,
                                     index=entity_names,
                                     columns=entity_names)
common_hypernym_frame
```



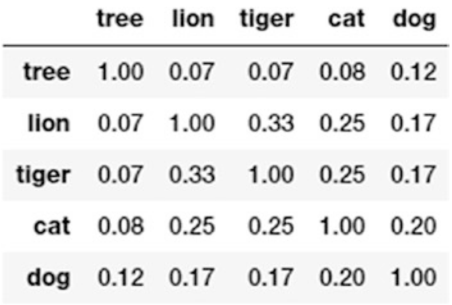
	tree	lion	tiger	cat	dog
tree	tree	organism	organism	organism	organism
lion	organism	lion	big_cat	feline	carnivore
tiger	organism	big_cat	tiger	feline	carnivore
cat	organism	feline	feline	cat	carnivore
dog	organism	carnivore	carnivore	carnivore	dog

Figure 8-2. *Pairwise common hypernym matrix*

Ignoring the main diagonal of the matrix, for each pair of entities, we can see their lowest common hypernym that depicts the nature of relationship between them (see Figure 8-2). Trees are unrelated to the other animals except both being living organisms. Hence, we get the “organism” relationship among them. Cats are related to lions and tigers with respect to them being feline creatures and we can see the same in the output. Tigers and lions are connected to each other with the “big cat” relationship. Finally, we can see that dogs have the relationship of “carnivore” with the other animals since they all typically eat meat.

We can also measure the semantic similarity between these entities using various semantic concepts. We will use *path similarity,* which returns a value between `[0, 1]` based on the shortest path that connects two terms based on their hypernym/hyponym based taxonomy. The following snippet shows how to generate this similarity matrix (see Figure 8-3).

```
similarities = []
for entity in entities:
    # get pairwise similarities
    similarities.append([round(entity.path_similarity(compared_entity), 2)
                         for compared_entity in entities])

# build pairwise similarity matrix
similarity_frame = pd.DataFrame(similarities, index=entity_names,
                                columns=entity_names)
similarity_frame
```



	tree	lion	tiger	cat	dog
tree	1.00	0.07	0.07	0.08	0.12
lion	0.07	1.00	0.33	0.25	0.17
tiger	0.07	0.33	1.00	0.25	0.17
cat	0.08	0.25	0.25	1.00	0.20
dog	0.12	0.17	0.17	0.20	1.00

Figure 8-3. *Pairwise similarity matrix*

From the output in Figure 8-3, as expected, lion and tiger are the most similar with a value of 0.33, followed by their semantic similarity with cat, having a value of 0.25. Tree has the lowest semantic similarity values when compared to the other animals. This concludes our discussion of analyzing lexical semantic relations. We encourage you explore more concepts with different examples by leveraging WordNet.

Word Sense Disambiguation

In the previous section, we looked at homographs and homonyms, which are basically words that look or sound similar but have very different meanings. This meaning is contextual based on how the word has been used and depends on the word semantics, also called *word sense.* Identifying the correct sense or semantics of a word based on its usage is called *word sense disambiguation,* with the assumption that the word has multiple meanings based on its context. This is a very popular problem in NLP and is used in various applications, like improving relevance of search engine results, coherence, and so on.

There are various ways to solve this problem, including lexical and dictionary based methods and supervised and unsupervised machine learning methods. Covering everything would be out of the current scope, hence we will be depicting word sense disambiguation using the Lesk algorithm, which is a classic algorithm invented by M. E. Lesk in 1986. The basic principle behind this algorithm is to leverage dictionary or vocabulary definitions for a word we want to disambiguate in a body of text and compare the words in these definitions with a section of text surrounding our word of interest. We will be using the WordNet definitions for words instead of a dictionary. The main objective is to return the synset with the maximum number of overlapping words or terms between the context sentence and the different definitions from each synset for the word we target for disambiguation. The following snippet leverages NLTK to depict how to use word sense disambiguation for various examples.

```
from nltk.wsd import lesk
from nltk import word_tokenize

# sample text and word to disambiguate
samples = [('The fruits on that plant have ripened', 'n'),
           ('He finally reaped the fruit of his hard work as he won the
           race', 'n')]

# perform word sense disambiguation
word = 'fruit'
for sentence, pos_tag in samples:
    word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
    print('Sentence:', sentence)
    print('Word synset:', word_syn)
    print('Corresponding definition:', word_syn.definition())
    print()
```

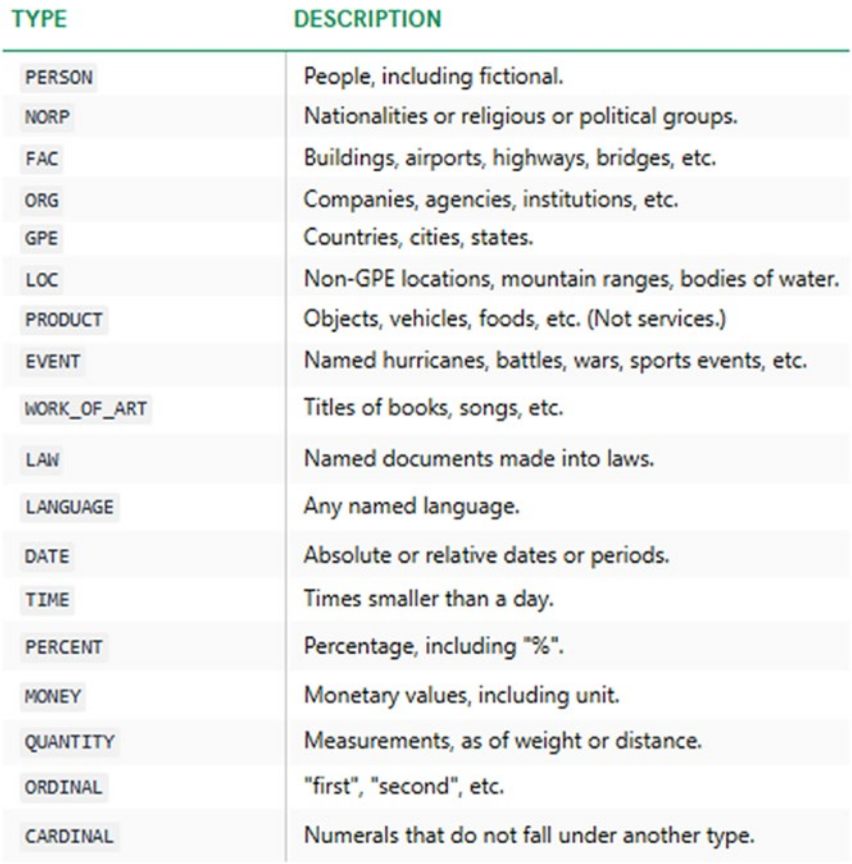
```
Sentence: The fruits on that plant have ripened
Word synset: Synset('fruit.n.01')
Corresponding definition: the ripened reproductive body of a seed plant
```

```
Sentence: He finally reaped the fruit of his hard work as he won the race
Word synset: Synset('fruit.n.03')
Corresponding definition: the consequence of some effort or action

# sample text and word to disambiguate
samples = [('Lead is a very soft, malleable metal', 'n'),
           ('John is the actor who plays the lead in that movie', 'n'),
           ('This road leads to nowhere', 'v')]

word = 'lead'


# perform word sense disambiguation
for sentence, pos_tag in samples:
    word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
    print('Sentence:', sentence)
    print('Word synset:', word_syn)
    print('Corresponding defition:', word_syn.definition())
    print()

Sentence: Lead is a very soft, malleable metal
Word synset: Synset('lead.n.02')
Corresponding definition: a soft heavy toxic malleable metallic element;
bluish white when freshly cut but tarnishes readily to dull grey

Sentence: John is the actor who plays the lead in that movie
Word synset: Synset('star.n.04')
Corresponding definition: an actor who plays a principal role

Sentence: This road leads to nowhere
Word synset: Synset('run.v.23')
Corresponding definition: cause something to pass or lead somewhere
```

We try to disambiguate two words—“fruit” and “lead”—in various text documents in these examples. You can see how we use the Lesk algorithm to get the correct word sense for the word we are disambiguating, based on its usage and context in each document. This tells you how “fruit” can mean both an entity that’s consumed as well as some consequence which one faces on applying efforts. We also see how “lead” can mean the soft metal, causing something/someone to go somewhere, or even an actor who plays the main role in a play or movie!

Named Entity Recognition

In any text document, there are particular terms that represent entities that are more informative and have a unique context compared to the rest of the text. These entities are known as *named entities,* and they more specifically represent real-world objects like people, places, organizations, and so on, which are usually denoted by proper names. We can find these by looking at the noun phrases in text documents. *Named entity recognition,* also called *entity chunking/extraction,* is a popular technique used in information extraction to identify and segment named entities and classify or categorize them under various predefined classes. SpaCy has some excellent capabilities for named entity recognition and you can find details on the general tagging scheme they use on their website at `https://spacy.io/api/annotation#named-entities`. We present the main named entity tags in the table depicted in Figure 8-4.



TYPE	DESCRIPTION
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.
LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

Figure 8-4. *Common named entities*

SpaCy offers a fast NER tagger based on a number of techniques. The exact algorithm hasn't been talked about in much detail, but the documentation marks it as "The exact algorithm is a pastiche of well-known methods, and is not currently described in any single publication". Let's try doing named entity recognition (NER) on a sample corpus now. For this, we define a sample news document as follows.

```
text = """Three more countries have joined an "international grand
committee" of parliaments, adding to calls for Facebook's boss, Mark
Zuckerberg, to give evidence on misinformation to the coalition. Brazil,
Latvia and Singapore bring the total to eight different parliaments across
the world, with plans to send representatives to London on 27 November
with the intention of hearing from Zuckerberg. Since the Cambridge
Analytica scandal broke, the Facebook chief has only appeared in front of
two legislatures: the American Senate and House of Representatives, and
the European parliament. Facebook has consistently rebuffed attempts from
others, including the UK and Canadian parliaments, to hear from Zuckerberg.
He added that an article in the New York Times on Thursday, in which the
paper alleged a pattern of behaviour from Facebook to "delay, deny and
deflect" negative news stories, "raises further questions about how recent
data breaches were allegedly dealt with within Facebook."

"""
```

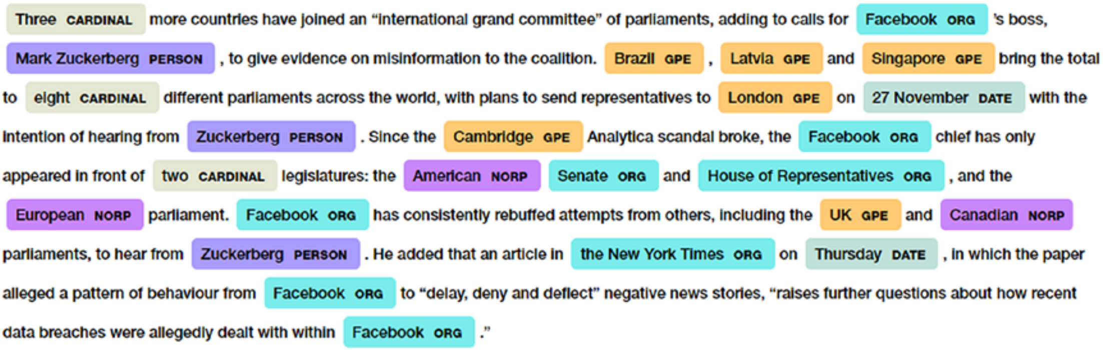
Getting the named entities is pretty easy now, thanks to spaCy. We do some basic text processing and obtained the NER tags using spaCy as follows.

```
import spacy
import re

text = re.sub(r'\n', ", text) # remove extra newlines
nlp = spacy.load('en')
text_nlp = nlp(text)
# print named entities in article
ner_tagged = [(word.text, word.ent_type_) for word in text_nlp]
print(ner_tagged)
```

```
[('Three', 'CARDINAL'), ('more', ''), ('countries', ''), ('have', ''),
('joined', ''), ('an', ''), ('''', ''), ('international', ''), ('grand', ''),
('committee', ''), ('"', ''), ('of', ''), ('parliaments', ''), (',', ''),
('adding', ''), ('to', ''), ('calls', ''), ('for', ''), ('Facebook', 'ORG'),
("s', ''), ('boss', ''), (',', ''), ('Mark', 'PERSON'), ('Zuckerberg', 'PERSON'),
(',', ''), ('to', ''), ('give', ''), ('evidence', ''), ('on', ''),
('misinformation', ''), ('to', ''), ('the', ''), ('coalition', ''), ('.', ''),
('Brazil', 'GPE'), (',', ''), ('Latvia', 'GPE'), ('and', ''), ('Singapore', 'GPE'),
('bring', ''), ('the', ''),('total', ''), ('to', ''), ('eight', 'CARDINAL'),
('different', ''), ('parliaments', ''), ('across', ''), ('the', ''),
('world', ''), (',', ''), ('with', ''), ('plans', ''), ('to', ''),
('send', ''), ('representatives', ''), ('to', ''), ('London', 'GPE'),
('on', ''), ('27', 'DATE'), ('November', 'DATE'), ('with', ''),
('the', ''), ('intention', ''), ('of', ''), ('hearing', ''), ('from', ''),
('Zuckerberg', 'PERSON'), ('.', ''), ('Since', ''), ('the', ''),
('Cambridge', 'GPE'), ('Analytica', ''), ('scandal', ''), ('broke', ''),
(',', ''), ('the', ''), ('Facebook', 'ORG'), ('chief', ''), ('has', ''),
('only', ''), ('appeared', ''), ('in', ''), ('front', ''), ('of', ''),
('two', 'CARDINAL'), ('legislatures', ''), (':', ''), ('the', ''),
('American', 'NORP'), ('Senate', 'ORG'), ('and', ''), ('House', 'ORG'),
('of', 'ORG'), ('Representatives', 'ORG'), (',', ''), ('and', ''),
('the', ''), ('European', 'NORP'), ('parliament', ''), ('.', ''),
('Facebook', 'ORG'), ('has', ''), ('consistently', ''), ('rebuffed', ''),
('attempts', ''), ('from', ''), ('others', ''), (',', ''), ('including', ''),
('the', ''), ('UK', 'GPE'), ('and', ''), ('Canadian', 'NORP'),
('parliaments', ''), (',', ''), ('to', ''), ('hear', ''), ('from', ''),
('Zuckerberg', 'PERSON'), ('.', ''), ('He', ''), ('added', ''), ('that', ''),
('an', ''), ('article', ''), ('in', ''), ('the', 'ORG'), ('New', 'ORG'),
('York', 'ORG'), ('Times', 'ORG'), ('on', ''), ('Thursday', 'DATE'), (',', ''),
('in', ''), ('which', ''), ('the', ''), ('paper', ''), ('alleged', ''),
('a', ''), ('pattern', ''), ('of', ''), ('behaviour', ''), ('from', ''),
('Facebook', 'ORG'), ('to', ''), ('''', ''), ('delay', ''), (',', ''),
('deny', ''), ('and', ''), ('deflect', ''), ('"', ''), ('negative', ''),
('news', ''), ('stories', ''), (',', ''), ('''', ''), ('raises', ''),
('further', ''), ('questions', ''), ('about', ''), ('how', ''), ('recent', ''),
('data', ''), ('breaches', ''), ('were', ''), ('allegedly', ''), ('dealt', ''),
('with', ''), ('within', ''), ('Facebook', 'ORG'), ('.', ''), ('"', '')]
```

You can clearly see several entities being identified in the text tokens in the preceding output. SpaCy also provides with a nice framework to view this in a visual manner as follows.

```
from spacy import displacy

# visualize named entities
displacy.render(text_nlp, style='ent', jupyter=True)
```



Three CARDINAL more countries have joined an “international grand committee” of parliaments, adding to calls for Facebook ORG 's boss,
Mark Zuckerberg PERSON , to give evidence on misinformation to the coalition. Brazil GPE , Latvia GPE and Singapore GPE bring the total
to eight CARDINAL different parliaments across the world, with plans to send representatives to London GPE on 27 November DATE with the
intention of hearing from Zuckerberg PERSON . Since the Cambridge GPE Analytica scandal broke, the Facebook ORG chief has only
appeared in front of two CARDINAL legislatures: the American NORP Senate ORG and House of Representatives ORG , and the
European NORP parliament. Facebook ORG has consistently rebuffed attempts from others, including the UK GPE and Canadian NORP
parliaments, to hear from Zuckerberg PERSON . He added that an article in the New York Times ORG on Thursday DATE , in which the paper
alleged a pattern of behaviour from Facebook ORG to “delay, deny and deflect” negative news stories, “raises further questions about how recent
data breaches were allegedly dealt with within Facebook ORG .”

Figure 8-5. *Named entities tagged by spaCy*

We can see all the major named entities tagged in the nice visualization depicted in Figure 8-5. Most of them make perfect sense, while some are slightly wrong. We can also programmatically extract the named entities, which is often more useful using the following code.

```
named_entities = []
temp_entity_name = "
temp_named_entity = None
for term, tag in ner_tagged:
    if tag:
        temp_entity_name = ' '.join([temp_entity_name, term]).strip()
        temp_named_entity = (temp_entity_name, tag)
    else:
        if temp_named_entity:
            named_entities.append(temp_named_entity)
            temp_entity_name = ''
            temp_named_entity = None
print(named_entities)
```

```
[('Three', 'CARDINAL'), ('Facebook', 'ORG'), ('Mark Zuckerberg', 'PERSON'),
('Brazil', 'GPE'), ('Latvia', 'GPE'), ('Singapore', 'GPE'), ('eight', 'CARDINAL'),
('London', 'GPE'), ('27 November', 'DATE'), ('Zuckerberg', 'PERSON'),
('Cambridge', 'GPE'), ('Facebook', 'ORG'), ('two', 'CARDINAL'),
('American Senate', 'ORG'), ('House of Representatives', 'ORG'),
('European', 'NORP'), ('Facebook', 'ORG'), ('UK', 'GPE'), ('Canadian', 'NORP'),
('Zuckerberg', 'PERSON'), ('the New York Times', 'ORG'), ('Thursday',
'DATE'), ('Facebook', 'ORG'), ('Facebook', 'ORG')]
```

```
# viewing the top entity types
from collections import Counter
c = Counter([item[1] for item in named_entities])
c.most_common()
```

```
[('ORG', 8), ('GPE', 6), ('CARDINAL', 3), ('PERSON', 3), ('DATE', 2),
('NORP', 2)]
```

This gives us a good idea of how to leverage spaCy for named entity recognition. Let's try to leverage the base Stanford NLP NER tagger now, using the relevant JAR files and the corresponding NLTK wrapper. Stanford's Named Entity Recognizer is based on an implementation of linear chain *Conditional Random Field* (CRF) sequence models. Prerequisites obviously include downloading the official Stanford NER Tagger JAR dependencies, which you can obtain at http://nlp.stanford.edu/software/stanford-ner-2014-08-27.zip. Or you can download the latest version from https://nlp.stanford.edu/software/CRF-NER.shtml#Download. Once it is downloaded, load it in NLTK as follows.

```
import os
from nltk.tag import StanfordNERTagger

JAVA_PATH = r'C:\Program Files\Java\jre1.8.0_192\bin\java.exe'
os.environ['JAVAHOME'] = JAVA_PATH
```

```
STANFORD_CLASSIFIER_PATH = 'E:/stanford/stanford-ner-2014-08-27/
classifiers/english.all.3class.distsim.crf.ser.gz'
STANFORD_NER_JAR_PATH = 'E:/stanford/stanford-ner-2014-08-27/stanford-ner.jar'

sn = StanfordNERTagger(STANFORD_CLASSIFIER_PATH,
                       path_to_jar=STANFORD_NER_JAR_PATH)
```

Now we can perform NER tagging and extract the relevant entities using the following code snippet.

```
text_enc = text.encode('ascii', errors='ignore').decode('utf-8')
ner_tagged = sn.tag(text_enc.split())

named_entities = []
temp_entity_name = ''
temp_named_entity = None
for term, tag in ner_tagged:
    if tag != 'O':
        temp_entity_name = ' '.join([temp_entity_name, term]).strip()
        temp_named_entity = (temp_entity_name, tag)
    else:
        if temp_named_entity:
            named_entities.append(temp_named_entity)
            temp_entity_name = ''
            temp_named_entity = None

print(named_entities)

[('Facebook', 'ORGANIZATION'), ('Latvia', 'LOCATION'), ('Singapore',
'LOCATION'), ('London', 'LOCATION'), ('Cambridge Analytica',
'ORGANIZATION'), ('Facebook', 'ORGANIZATION'), ('Senate', 'ORGANIZATION'),
('Facebook', 'ORGANIZATION'), ('UK', 'LOCATION'), ('New York Times',
'ORGANIZATION'), ('Facebook', 'ORGANIZATION')]

# get most frequent entities
c = Counter([item[1] for item in named_entities])
c.most_common()

[('ORGANIZATION', 7), ('LOCATION', 4)]
```

There is one limitation, however. This model is only trained on instances of PERSON, ORGANIZATION, *and* LOCATION types, which is kind of limiting compared to spaCy. Luckily, a newer version of Stanford Core NLP is available and the newer APIs in NLTK recommend using it. However, to use Stanford's Core NLP from NLTK in Python, we need to download and start a Core NLP server locally. Why do we need this? NLTK is slowly deprecating the old Stanford parsers in favor of the more active Stanford Core NLP project. It might even get removed after NLTK version 3.4, so best to stay updated. You can find out further details in this GitHub issue for NLTK at https://github.com/nltk/nltk/issues/1839.

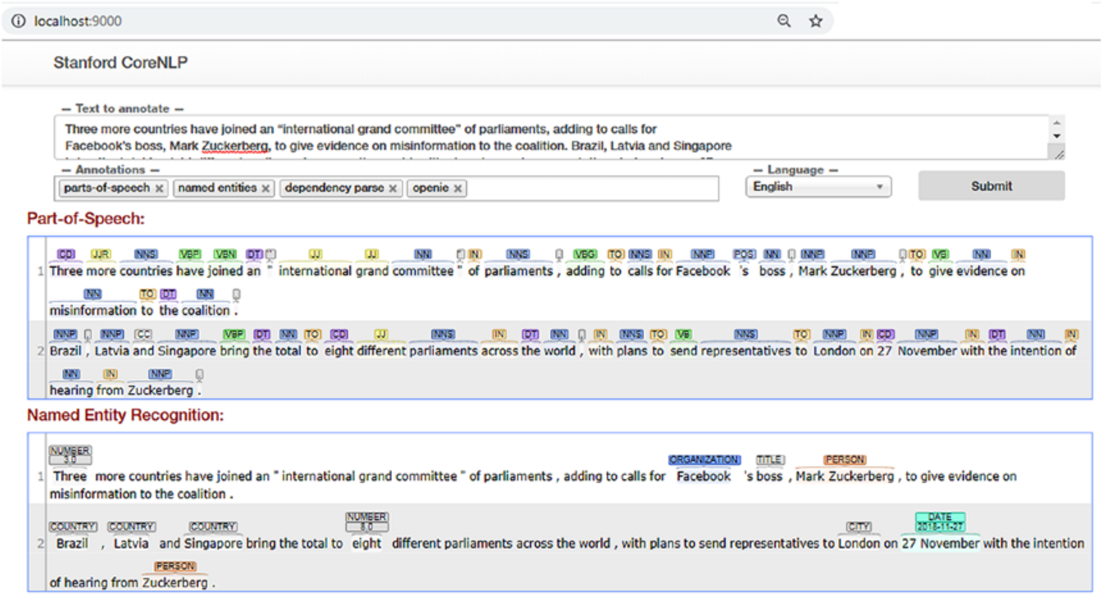
We will start by downloading Stanford's Core NLP suite from https://stanfordnlp.github.io/CoreNLP/. After you download and extract the directory, go there and start the Core NLP server using the following command from the terminal.

```
E:\> java -mx4g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer
-preload tokenize,ssplit,pos,lemma,ner,parse,depparse -status_port 9000
-port 9000 -timeout 15000
```

If it runs successfully, you should see the following messages on the terminal when it starts up.

```
E:\stanford\stanford-corenlp-full-2018-02-27>java -mx4g -cp "*" edu.
stanford.nlp.pipeline.StanfordCoreNLPServer -preload tokenize,ssplit,pos,le
mma,ner,parse,depparse -status_port 9000 -port 9000 -timeout 15000
[main] INFO CoreNLP - --- StanfordCoreNLPServer#main() called ---
...
...
[main] INFO edu.stanford.nlp.pipeline.TokensRegexNERAnnotator -
TokensRegexNERAnnotator ner.fine.regexner: Read 580641 unique entries
out of 581790 from edu/stanford/nlp/models/kbp/regexner_caseless.tab, 0
TokensRegex patterns.
...
...
[main] INFO CoreNLP - Starting server...
[main] INFO CoreNLP - StanfordCoreNLPServer listening at
/0:0:0:0:0:0:0:0:9000
```

If you're interested, you can head over to http://localhost:9000 and play around with their intuitive user interface, as depicted in Figure 8-6.



Three more countries have joined an "international grand committee" of parliaments, adding to calls for Facebook's boss, Mark Zuckerberg, to give evidence on misinformation to the coalition. Brazil, Latvia and Singapore

Part-of-Speech:

1 Three more countries have joined an " international grand committee " of parliaments , adding to calls for Facebook 's boss , Mark Zuckerberg , to give evidence on misinformation to the coalition .

2 Brazil , Latvia and Singapore bring the total to eight different parliaments across the world , with plans to send representatives to London on 27 November with the intention of hearing from Zuckerberg .

Named Entity Recognition:

1 Three more countries have joined an " international grand committee " of parliaments , adding to calls for Facebook 's boss , Mark Zuckerberg , to give evidence on misinformation to the coalition .

2 Brazil , Latvia and Singapore bring the total to eight different parliaments across the world , with plans to send representatives to London on 27 November with the intention of hearing from Zuckerberg .

Figure 8-6. *Exploring Stanford Core NLP*

You can visualize text annotations and tags interactively using the user interface depicted in Figure 8-6. We will now use it in Python through NLTK as follows.

```
from nltk.parse import CoreNLPParser
import nltk

# NER Tagging
ner_tagger = CoreNLPParser(url='http://localhost:9000', tagtype='ner')
tags = list(ner_tagger.raw_tag_sents(nltk.sent_tokenize(text)))
tags = [sublist[0] for sublist in tags]
tags = [word_tag for sublist in tags for word_tag in sublist]

# Extract Named Entities
named_entities = []
temp_entity_name = "
temp_named_entity = None
for term, tag in tags:
    if tag != 'O':
        temp_entity_name = ' '.join([temp_entity_name, term]).strip()
        temp_named_entity = (temp_entity_name, tag)
    else:
        if temp_named_entity:
            named_entities.append(temp_named_entity)
            temp_entity_name = "
            temp_named_entity = None
```

```
print(named_entities)
```

```
[('Three', 'NUMBER'), ('Facebook', 'ORGANIZATION'), ('boss',
'TITLE'), ('Mark Zuckerberg', 'PERSON'), ('Brazil', 'COUNTRY'),
('Latvia', 'COUNTRY'), ('Singapore', 'COUNTRY'), ('eight', 'NUMBER'),
('London', 'CITY'), ('27 November', 'DATE'), ('Zuckerberg', 'PERSON'),
('Cambridge Analytica', 'ORGANIZATION'), ('Facebook', 'ORGANIZATION'),
('two', 'NUMBER'), ('American Senate', 'ORGANIZATION'), ('House of
Representatives', 'ORGANIZATION'), ('European', 'NATIONALITY'),
('Facebook', 'ORGANIZATION'), ('UK', 'COUNTRY'), ('Canadian',
'NATIONALITY'), ('Zuckerberg', 'PERSON'), ('New York Times',
'ORGANIZATION'), ('Thursday', 'DATE'), ('Facebook', 'ORGANIZATION'),
('Facebook', 'ORGANIZATION')]
```

```
# Find out top named entity types
c = Counter([item[1] for item in named_entities])
c.most_common()
```

```
[('ORGANIZATION', 9), ('COUNTRY', 4), ('NUMBER', 3), ('PERSON', 3),
 ('DATE', 2), ('NATIONALITY', 2), ('TITLE', 1), ('CITY', 1)]
```

Thus you can see that Core NLP has many more tag types compared to the previous version. In fact, it recognizes named entities (PERSON, LOCATION, ORGANIZATION, and MISC), numerical entities (MONEY, NUMBER, ORDINAL, and PERCENT), and temporal entities (DATE, TIME, DURATION, and SET). There are 12 classes in all.

Building an NER Tagger from Scratch

There are various off-the-shelf solutions that offer capabilities to perform named entity extraction (some of which we discussed in the previous sections). Yet there are times when the requirements are beyond the capabilities of off-the-shelf classifiers. In this section, we go through an exercise to build our own NER using conditional random fields. We use a popular framework called sklearn_crfsuite to develop our NER.

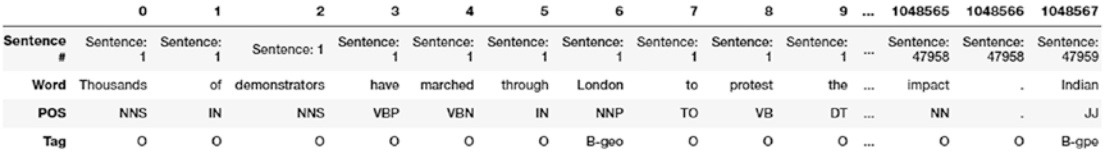
The key point to remember here is that NER is a sequence modeling problem at its core. It is more related to the classification suite of problems, wherein we need a labeled dataset to train a classifier. Without any training data, there is no NER model! There are various labeled datasets for NER class of problems. We utilize a preprocessed version of the GMB (Groningen Meaning Bank) corpus for this tutorial. The preprocessed version is available at https://www.kaggle.com/abhinavwalia95/entity-annotated-corpus. However, we also provide it in our GitHub repository at https://github.com/dipanjanS/text-analytics-with-python for ease of use. Loading the dataset, we can check the major fields as follows.

```
import pandas as pd

df = pd.read_csv('ner_dataset.csv.gz', compression='gzip',
encoding='ISO-8859-1')
df = df.fillna(method='ffill')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1048575 entries, 0 to 1048574
Data columns (total 4 columns):
Sentence #    1048575 non-null object
Word          1048575 non-null object
POS           1048575 non-null object
Tag           1048575 non-null object
dtypes: object(4)
memory usage: 32.0+ MB
```

We can also take a look at the actual dataset by using the following code. See Figure 8-7.

```
df.T
```



	0	1	2	3	4	5	6	7	8	9	...	1048565	1048566	1048567
Sentence #	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	Sentence: 1	...	Sentence: 47958	Sentence: 47958	Sentence: 47959
Word	Thousands	of	demonstrators	have	marched	through	London	to	protest	the	...	impact	.	Indian
POS	NNS	IN	NNS	VBP	VBN	IN	NNP	TO	VB	DT	...	NN	.	JJ
Tag	O	O	O	O	O	O	B-geo	O	O	O	...	O	O	B-gpe

***Figure 8-7.** The GMB NER dataset*

To get a deeper understanding of the data we are dealing with and the total number of annotated tags, we can use the following code.

```
df['Sentence #'].nunique(), df.Word.nunique(), df.POS.nunique(), df.Tag.
nunique()
```

```
(47959, 35178, 42, 17)
```

This tells us that we have 47,959 sentences that contain 35,178 unique words. These sentences have a total of 42 unique POS tags and 17 unique NER tags. We can check out the unique NER tag distribution in our corpus as follows.

```
df.Tag.value_counts()
```

```
O        887908
B-geo     37644
B-tim     20333
B-org     20143
I-per     17251
B-per     16990
I-org     16784
B-gpe     15870
I-geo      7414
I-tim      6528
B-art       402
B-eve       308
I-art       297
I-eve       253
B-nat       201
I-gpe       198
I-nat        51
Name: Tag, dtype: int64
```

The preceding output shows the unbalanced distribution of different tags in the dataset. The GMB dataset utilizes *IOB tagging* (Inside, Outside Beginning). IOB is a common tagging format for tagging tokens, which we discussed in Chapter 3. To refresh your memory:

- `I-` prefix before a tag indicates that the tag is inside a chunk.
- `B-` prefix before a tag indicates that the tag is the beginning of a chunk.
- `O-` tag indicates that a token belongs to no chunk (outside).

The NER tags in this dataset can be explained using the following notation, which is similar to the NER tags you have seen so far.

- `geo` = Geographical entity
- `org` = Organization
- `per` = Person
- `gpe` = Geopolitical entity
- `tim` = Time indicator
- `art` = Artifact
- `eve` = Event
- `nat` = Natural phenomenon

Anything outside these classes is called *other*, denoted as `O`. Now, as mentioned earlier, NER belongs to the sequence modeling class of problems. There are different algorithms to tackle sequence modeling, and CRF (Conditional Random Fields) is one such example. CRFs are proven to perform extremely well on NER and related domains. In this tutorial, we will attempt to develop our own NER based on CRFs. Discussion CRFs in detail are beyond the scope given this is not a hardcore machine learning book. To whet your appetite though, a CRF is an undirected graphical model whose nodes can be divided into exactly two disjoint sets ***X*** and ***Y***, the observed and output variables, respectively; the conditional distribution ***p(Y | X)*** is then modeled. We recommend interested readers check out the following literature on CRFs to gain a deep dive: `https://repository.upenn.edu/cgi/viewcontent.cgi?article=1162&context=cis_papers`.

Feature engineering is critical for building any machine learning or statistical model because without features, there is no learning. Similarly, the CRF model trains sequences of input features to learn transitions from one state (label) to another. To enable such an algorithm, we need to define features, which take into account different transitions. We will develop a function called `word2features()`, where we will transform each word into a feature dictionary depicting the following attributes or features:

- Lowercase version of the word
- Suffix containing the last three characters
- Suffix containing the last two characters
- Flags to determine uppercase, title case, numeric data, and POS tags

We also attach attributes related to previous and next words or tags to determine beginning of sentence (BOS) or end of sentence (EOS). These are features based on best practices and you can add your own features with experimentation. Always remember that feature engineering is an art as well as a science.

```
def word2features(sent, i):
    word = sent[i][0]
    postag = sent[i][1]

    features = {
        'bias': 1.0,
        'word.lower()': word.lower(),
        'word[-3:]': word[-3:],
        'word[-2:]': word[-2:],
        'word.isupper()': word.isupper(),
        'word.istitle()': word.istitle(),
        'word.isdigit()': word.isdigit(),
        'postag': postag,
        'postag[:2]': postag[:2],
    }
    if i > 0:
        word1 = sent[i-1][0]
        postag1 = sent[i-1][1]
        features.update({
            '-1:word.lower()': word1.lower(),
            '-1:word.istitle()': word1.istitle(),
            '-1:word.isupper()': word1.isupper(),
            '-1:postag': postag1,
            '-1:postag[:2]': postag1[:2],
        })
    else:
        features['BOS'] = True
    if i < len(sent)-1:
        word1 = sent[i+1][0]
        postag1 = sent[i+1][1]
        features.update({
            '+1:word.lower()': word1.lower(),
            '+1:word.istitle()': word1.istitle(),
            '+1:word.isupper()': word1.isupper(),
            '+1:postag': postag1,
            '+1:postag[:2]': postag1[:2],
        })
    else:
        features['EOS'] = True

    return features

# convert input sentence into features
def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]
# get corresponding outcome NER tag label for input sentence
def sent2labels(sent):
    return [label for token, postag, label in sent]
```

Let’s now define a function to extract our word token, POS tag, and NER tag triplets from sentences. We will be applying this to all our input sentences.

```
agg_func = lambda s: [(w, p, t) for w, p, t in zip(s['Word'].values.tolist(),
                                                  s['POS'].values.tolist(),
                                                  s['Tag'].values.tolist())]
grouped_df = df.groupby('Sentence #').apply(agg_func)
```

We can now view a sample annotated sentence from our dataset with the following code.

```
sentences = [s for s in grouped_df]
sentences[0]
```

```
('of', 'IN', 'O'), ('demonstrators', 'NNS', 'O'), ('have', 'VBP', 'O'),
('marched', 'VBN', 'O'), ('through', 'IN', 'O'), ('London', 'NNP', 'B-geo'),
('to', 'TO', 'O'), ('protest', 'VB', 'O'), ('the', 'DT', 'O'),
('war', 'NN', 'O'), ('in', 'IN', 'O'), ('Iraq', 'NNP', 'B-geo'), ('and', 'CC', 'O'),
('demand', 'VB', 'O'),('the', 'DT', 'O'), ('withdrawal', 'NN', 'O'),
('of', 'IN', 'O'), ('British', 'JJ', 'B-gpe'), ('troops', 'NNS', 'O'),
('from', 'IN', 'O'), ('that', 'DT', 'O'), ('country', 'NN', 'O'), ('.', '.', 'O')]
```

The preceding output shows a standard tokenized sentence with POS and NER tags. Let's look at how each annotated tokenized sentence can be used for our feature engineering with the function we defined earlier.

```
sent2features(sentences[0][5:7])
```

```
[{'bias': 1.0, 'word.lower()': 'through', 'word[-3:]': 'ugh', 'word[-2:]': 'gh',
  'word.isupper()': False, 'word.istitle()': False, 'word.isdigit()': False,
  'postag': 'IN', 'postag[:2]': 'IN', 'BOS': True, '+1:word.lower()': 'london',
  '+1:word.istitle()': True, '+1:word.isupper()': False, '+1:postag': 'NNP',
  '+1:postag[:2]': 'NN'},
 {'bias': 1.0,  'word.lower()': 'london', 'word[-3:]': 'don', 'word[-2:]': 'on',
  'word.isupper()': False, 'word.istitle()': True, 'word.isdigit()': False,
  'postag': 'NNP', 'postag[:2]': 'NN', '-1:word.lower()': 'through',
  '-1:word.istitle()': False, '-1:word.isupper()': False, '-1:postag': 'IN',
  '-1:postag[:2]': 'IN', 'EOS': True}]
```

```
sent2labels(sentences[0][5:7])
```

```
['O', 'B-geo']
```

The preceding output shows features on two sample word tokens and their corresponding NER tag labels. Let's now prepare our train and test datasets by feature engineering on the input sentences and getting the corresponding NER tag labels to predict.

```
from sklearn.model_selection import train_test_split
import numpy as np

X = np.array([sent2features(s) for s in sentences])
y = np.array([sent2labels(s) for s in sentences])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
X_train.shape, X_test.shape
```

```
((35969,), (11990,))
```

It is now time to start training our model. For this, we use `sklearn-crfsuite` like we mentioned before. The `sklearn-crfsuite` framework is a thin CRFsuite (`python-crfsuite`) wrapper that provides a Scikit-Learn compatible `sklearn_crfsuite.CRF estimator`. Thus, you can use Scikit-Learn model selection utilities (cross-validation and hyperparameter optimization) with it and save/load CRF models using `joblib`. You can install the library using the `pip install sklearn_crfsuite` command.

We will now train the model using the default configurations mentioned in the `sklearn-crfsuite` API docs, which you can access at `https://sklearn-crfsuite.readthedocs.io/en/latest/api.html`. The intent here is NER tagging, so we will not focus too much on tuning our model. Some key hyperparameters and model arguments are mentioned here:

- `algorithm`: The training algorithm. We use `L-BFGS` for gradient descent for optimization and getting model parameters
- `c1`: Coefficient for Lasso (L1) regularization
- `c2`: Coefficient for Ridge (L2) regularization
- `all_possible_transitions`: Specify whether CRFsuite generates transition features that do not occur in the training data

```
import sklearn_crfsuite

crf = sklearn_crfsuite.CRF(algorithm='lbfgs',
                           c1=0.1,
                           c2=0.1,
                           max_iterations=100,
                           all_possible_transitions=True,
                           verbose=True)
crf.fit(X_train, y_train)
```

```
loading training data to CRFsuite: 100%|█████| 35969/35969 [00:15<00:00,
2384.94it/s]
type: CRF1d
Number of features: 133629
Seconds required: 3.486

L-BFGS optimization
c1: 0.100000
c2: 0.100000
num_memories: 6
max_iterations: 100
epsilon: 0.000010
stop: 10
delta: 0.000010
linesearch: MoreThuente
linesearch.max_iterations: 20

Iter 1   time=4.01  loss=1264028.26 active=132637 feature_norm=1.00
Iter 2   time=3.99  loss=994059.01 active=131294 feature_norm=4.42
...
...
Iter 99  time=2.07  loss=32324.92 active=58249 feature_norm=219.98
Iter 100 time=2.09  loss=32316.67 active=58226 feature_norm=220.04
L-BFGS terminated with the maximum number of iterations
Total seconds required for training: 228.530

Storing the model
Number of active features: 58226 (133629)
Number of active attributes: 29279 (90250)
Number of active labels: 17 (17)
```

You can now save this model using the following code, which leverages the `joblib` framework.

```
from sklearn.externals import joblib

joblib.dump(crf, 'ner_model.pkl')
```

If this model is taking too long to train, you can load the pretrained model provided in our GitHub repository at https://github.com/dipanjanS/text-analytics-with-python using the following code.

```
crf = joblib.load('ner_model.pkl')
```

Let's evaluate our model performance for NER tagging on the test data now! The following code shows a sample prediction and the actual labels. Looks like we are doing well!

```
y_pred = crf.predict(X_test)
print(y_pred[0])

['O', 'O', 'O', 'O', 'B-per', 'I-per', 'O', 'B-org', 'O', 'O', 'B-gpe',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O']

print(y_test[0])

['O', 'O', 'O', 'O', 'B-per', 'I-per', 'O', 'B-org', 'O', 'O', 'B-gpe',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O']
```

The following code helps us evaluate our model performance on the entire test dataset and get key classification model performance metrics.

```
from sklearn_crfsuite import metrics as crf_metrics

labels = list(crf.classes_)
labels.remove('O')
print(crf_metrics.flat_classification_report(y_test, y_pred,
labels=labels))
```

```
              precision    recall  f1-score   support

       B-org       0.81      0.73      0.77      5116
       B-per       0.85      0.84      0.84      4239
       I-per       0.85      0.90      0.88      4273
       B-geo       0.86      0.91      0.89      9403
       I-geo       0.81      0.80      0.81      1826
       B-tim       0.93      0.89      0.91      5095
       I-org       0.82      0.79      0.80      4195
       B-gpe       0.97      0.94      0.96      3961
       I-tim       0.84      0.81      0.82      1604
       B-nat       0.50      0.24      0.32        55
       B-eve       0.51      0.33      0.40        80
       B-art       0.36      0.14      0.20       102
       I-art       0.24      0.07      0.10        90
       I-eve       0.45      0.19      0.27        74
       I-gpe       0.86      0.53      0.66        36
       I-nat       0.57      0.22      0.32        18

   micro avg       0.86      0.85      0.86     40167
   macro avg       0.70      0.58      0.62     40167
weighted avg       0.86      0.85      0.85     40167
```

We have intentionally left out the Others tag to understand the performance of the model on the remaining tags, which are of key interest. The evaluation statistics showcase a model that seems to have learned the transitions quite well, giving us an overall F1-score of 85%! We can achieve even better results by fine tuning the feature engineering step along with hyper-parameter tuning.

Building an End-to-End NER Tagger with Our Trained NER Model

There is no fun (or value!) if we cannot use our model to tag new sentences in the future, assuming we would want to put this model in production. Let's try to build an end-to-end workflow to perform NER tagging on our sample document. Just to refresh your memory, our sample document is as follows.

```
import re

text = """Three more countries have joined an "international grand
committee" of parliaments, adding to calls forFacebook’s boss, Mark
Zuckerberg, to give evidence on misinformation to the coalition. Brazil,
Latvia and Singapore bring the total to eight different parliaments across
the world, with plans to send representatives to London on 27 November
with the intention of hearing from Zuckerberg. Since the Cambridge
Analytica scandal broke, the Facebook chief has only appeared in front of
two legislatures: the American Senate and House of Representatives, and
the European parliament. Facebook has consistently rebuffed attempts from
others, including the UK and Canadian parliaments, to hear from Zuckerberg.
He added that an article in the New York Times on Thursday, in which the
paper alleged a pattern of behaviour from Facebook to "delay, deny and
deflect" negative news stories, "raises further questions about how recent
data breaches were allegedly dealt with within Facebook."
"""
text = re.sub(r'\n', ", text)
```

The first step in the pipeline is to tokenize our text and perform POS tagging, as depicted in the following code.

```
import nltk

text_tokens = nltk.word_tokenize(text)
text_pos = nltk.pos_tag(text_tokens)
text_pos[:10]
```

```
[('Three', 'CD'), ('more', 'JJR'), ('countries', 'NNS'), ('have', 'VBP'),
('joined', 'VBN'), ('an', 'DT'), ('"', 'NNP'), ('international', 'JJ'),
('grand', 'JJ'), ('committee', 'NN')]
```

The next step is to extract features from the POS tagged text document, which we can do using our previously defined function.

```
features = [sent2features(text_pos)]
features[0][0]
```

```
{'bias': 1.0, 'word.lower()': 'three', 'word[-3:]': 'ree', 'word[-2:]': 'ee',
 'word.isupper()': False, 'word.istitle()': True, 'word.isdigit()': False,
 'postag': 'CD', 'postag[:2]': 'CD', 'BOS': True, '+1:word.lower()': 'more',
 '+1:word.istitle()': False, '+1:word.isupper()': False, '+1:postag': 'JJR',
 '+1:postag[:2]': 'JJ'}
```

It is now time to use the CRF model we just trained to predict the features we engineered from our sample document.

```
labels = crf.predict(features)
doc_labels = labels[0]
doc_labels[10:20]
```

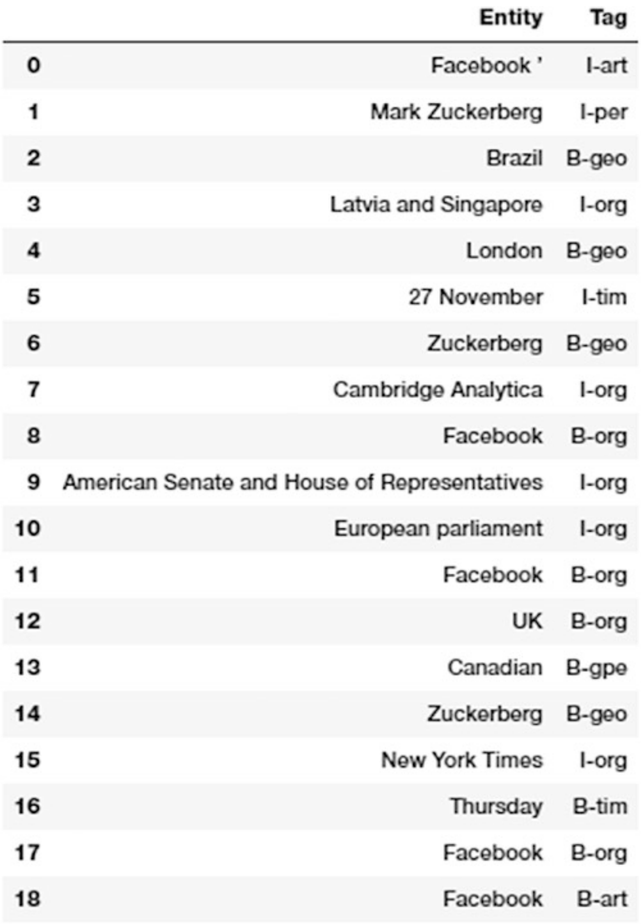
```
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-art', 'I-art']
```

The final step involves combining the actual text tokens with their corresponding NER tags and retrieving relevant named entities from the NER tags. See Figure 8-8.

```
text_ner = [(token, tag) for token, tag in zip(text_tokens, doc_labels)]
print(text_ner)
```

```
[('Three', 'O'), ('more', 'O'), ('countries', 'O'), ..., ('Facebook',
'B-art'), ("', 'I-art'), ('s', 'O'), ('boss', 'O'), (',', 'O'), ('Mark',
'B-per'), ('Zuckerberg', 'I-per'), (',', 'O'), ('to', 'O'), ('give', 'O'),
('evidence', 'O'), ('on', 'O'), ('misinformation', 'O'), ('to', 'O'),
('the', 'O'), ('coalition', 'O'), ('.', 'O'), ('Brazil', 'B-geo'), ...]
```

```
# extract and display all named entities
named_entities = []
temp_entity_name = "
temp_named_entity = None
for term, tag in text_ner:
    if tag != 'O':
        temp_entity_name = ' '.join([temp_entity_name, term]).strip()
        temp_named_entity = (temp_entity_name, tag)
    else:
        if temp_named_entity:
            named_entities.append(temp_named_entity)
            temp_entity_name = "
            temp_named_entity = None

import pandas as pd
pd.DataFrame(named_entities, columns=['Entity', 'Tag'])
```



	Entity	Tag
0	Facebook '	I-art
1	Mark Zuckerberg	I-per
2	Brazil	B-geo
3	Latvia and Singapore	I-org
4	London	B-geo
5	27 November	I-tim
6	Zuckerberg	B-geo
7	Cambridge Analytica	I-org
8	Facebook	B-org
9	American Senate and House of Representatives	I-org
10	European parliament	I-org
11	Facebook	B-org
12	UK	B-org
13	Canadian	B-gpe
14	Zuckerberg	B-geo
15	New York Times	I-org
16	Thursday	B-tim
17	Facebook	B-org
18	Facebook	B-art

Figure 8-8. *Named entities from our NER model*

Congratulations! You have built your own NER tagger from scratch and it is performing quite well. This should enable you to build more sophisticated NER taggers, maybe in your own specialized domains.

Analyzing Semantic Representations

We usually communicate in the form of messages, either in spoken form or written form, with other people or interfaces. These messages are typically a collection of words, phrases, and sentences. They have their own semantics and context. So far we talked about semantics and relations between various lexical units. But how do we represent the meaning of semantics conveyed by a message or messages? How do humans understand what someone is telling them? How do we believe in statements and propositions and evaluate outcomes and what action to take? It is easy because the brain helps us in logic and reasoning, but computationally can we do the same? The answer is yes we can. Frameworks like propositional logic and first order logic help us in the representation of semantics. We discussed this in detail in Chapter 1, in the subsection "Representation of Semantics" under the "Language Semantics" section. We encourage you to go through that section once more to refresh your memory. In the following sections, we look at ways to represent propositional and first order logic to prove or disprove propositions, statements, and predicates using practical examples and code.

Propositional Logic

We already explained that propositional logic is the study of propositions, statements, and sentences. A proposition is usually declarative, having a binary value of either true or false. There are also various logical operators like conjunction, disjunction, implication, and equivalence and we also study the effects of applying these operators to multiple propositions to understand their behavior and outcome. Let's consider our example from Chapter 1 with regards to two propositions `P` and `Q`, such that they can be represented as follows.

P: He is hungry

Q: He will eat a sandwich

We will now try to build truth tables for operations on these propositions using NLTK based on the various logical operators discussed in Chapter 1 (refer to the "Propositional Logic" section for more details) and then derive outcomes computationally.

```
import nltk
import pandas as pd
import os
```

```
# assign symbols and propositions
symbol_P = 'P'
symbol_Q = 'Q'

proposition_P = 'He is hungry'
propositon_Q = 'He will eat a sandwich'

# assign various truth values to the propositions
p_statuses = [False, False, True, True]
q_statuses = [False, True, False, True]

# assign the various expressions combining the logical operators
conjunction = '(P & Q)'
disjunction = '(P | Q)'
implication = '(P -> Q)'
equivalence = '(P <-> Q)'
expressions = [conjunction, disjunction, implication, equivalence]
expressions

['(P & Q)', '(P | Q)', '(P -> Q)', '(P <-> Q)']

# evaluate each expression using propositional logic
results = []
for status_p, status_q in zip(p_statuses, q_statuses):
    dom = set([])
    val = nltk.Valuation([(symbol_P, status_p),
                          (symbol_Q, status_q)])
    assignments = nltk.Assignment(dom)
    model = nltk.Model(dom, val)
    row = [status_p, status_q]
    for expression in expressions:
      # evaluate each expression based on proposition truth values
        result = model.evaluate(expression, assignments)
        row.append(result)
    results.append(row)

# build the result table
columns = [symbol_P, symbol_Q, conjunction,
           disjunction, implication, equivalence]
result_frame = pd.DataFrame(results, columns=columns)

# display results
print('P:', proposition_P)
print('Q:', propositon_Q)
print()
print('Expression Outcomes:-')
print(result_frame)
```

```
P: He is hungry
Q: He will eat a sandwich

Expression Outcomes:-
       P      Q (P & Q) (P | Q) (P -> Q) (P <-> Q)
0  False  False   False   False     True      True
1  False   True   False    True     True     False
2   True  False   False    True    False     False
3   True   True    True    True     True      True
```

This output depicts the various truth values of the two propositions. When we combine them with various logical operators, you will find that the results match what we manually evaluated in Chapter 1. For example, **P & Q** indicates that "he is hungry and he will eat a sandwich" is true only when both of the individual propositions are true. We use NLTK's `Valuation` class to create a dictionary of the propositions and their various outcome states. We use the `Model` class to evaluate each expression, where the `evaluate()` function internally calls the recursive function `satisfy()`, which helps to evaluate the outcome of each expression with the propositions based on the assigned truth values.

First Order Logic

Propositional logic (PL) has several limitations, like the inability to represent facts or complex relationships and inferences. PL also has limited expressive power because, for each new proposition, we need a unique symbolic representation and it becomes very difficult to generalize facts. This is where first order logic (FOL) works really well with features like functions, quantifiers, relations, connectives, and symbols. It provides a more rich and powerful representation for semantic information. The "First Order

Logic" subsection under "Representation of Semantics" in Chapter 1 provides detailed information about how first order logic works.

In this section, we build several FOL representations similar to what we did manually in Chapter 1 using mathematical representations. Here, we build them in our code using similar syntax and leverage NLTK and some theorem provers to prove the outcomes of various expressions based on predefined conditions and relationships, similar to what we did for PL.

The key takeaway for you from this section should be getting to know how to represent FOL representations in Python and how to perform first order logic inferences using proofs based on some goal and predefined rules and events. There are several theorem provers that you can use to evaluate expressions and proving theorems. The NLTK package has three provers, namely `Prover9`, `TableauProver`, and `ResolutionProver`. The first one is free and available for download at `https://www.cs.unm.edu/~mccune/prover9/download/`. You can extract the contents in a location of your choice. We use both `ResolutionProver` and `Prover9` in our examples. The following snippet helps set up the necessary dependencies for FOL expressions and evaluations.

```
import nltk
import os

# for reading FOL expressions
read_expr = nltk.sem.Expression.fromstring

# initialize theorem provers (you can choose any)
os.environ['PROVER9'] = r'E:/prover9/bin'
prover = nltk.Prover9()

# I use the following one for our examples
prover = nltk.ResolutionProver()
```

Now that we have our dependencies ready, let's evaluate a few FOL expressions. Consider a simple expression that says "If an entity jumps over another entity, the reverse cannot happen". Assuming the entities to be **x** and **y**, we can represent this is FOL as ∀x ∀y `(jumps_over(x, y)` → ¬`jumps_over(y, x))`, which signifies that for all x and y, if x jumps over y, it implies that y cannot jump over x. Consider now that we have two entities—`fox` and `dog`—and the `fox` jumps over the `dog`. This event can be represented by `jumps_over(fox, dog)`. Our objective is to evaluate the outcome of jumps_over(dog, fox) considering this expression and the event that occurred. The following snippet shows how we can do this.

```
# set the rule expression
rule = read_expr('all x. all y. (jumps_over(x, y) -> -jumps_over(y, x))')

# set the event occurred
event = read_expr('jumps_over(fox, dog)')

# set the outcome we want to evaluate -- the goal
test_outcome = read_expr('jumps_over(dog, fox)')

# get the result
prover.prove(goal=test_outcome,
             assumptions=[event, rule],
             verbose=True)

[1] {-jumps_over(dog,fox)}                    A
[2] {jumps_over(fox,dog)}                     A
[3] {-jumps_over(z4,z3), -jumps_over(z3,z4)}  A
[4] {-jumps_over(dog,fox)}                    (2, 3)

Out[9]: False
```

This output depicts the final result for our goal test_outcome is false, i.e., the dog cannot jump over the fox if the fox has already jumped over the dog. This is based on our rule expression and the events that are assigned to the assumptions parameter in the prover. The sequence of steps that lead to the result is also shown in the output.

Let's now consider another FOL expression rule: ∀x studies(x, exam) → pass(x, exam). This tells us that for all instances of x, if x studies for the exam, he/she will pass the exam. Let's represent this rule and consider two students—John and Pierre—and assume that John does not study for the exam, but Pierre does. Can we then determine whether they will pass the exam based on the expression rule? The following snippet shows the result.

```
# set the rule expression
rule = read_expr('all x. (studies(x, exam) -> pass(x, exam))')

# set the events and outcomes we want to determine
event1 = read_expr('-studies(John, exam)')
```

```
test_outcome1 = read_expr('pass(John, exam)')

# get results
prover.prove(goal=test_outcome1,
             assumptions=[event1, rule],
             verbose=True)

[1] {-pass(John,exam)}                  A
[2] {-studies(John,exam)}               A
[3] {-studies(z6,exam), pass(z6,exam)}  A
[4] {-studies(John,exam)}               (1, 3)

Out[10]: False

# set the events and outcomes we want to determine
event2 = read_expr('studies(Pierre, exam)')
test_outcome2 = read_expr('pass(Pierre, exam)')

# get results
prover.prove(goal=test_outcome2,
             assumptions=[event2, rule],
             verbose=True)

[1] {-pass(Pierre,exam)}                A
[2] {studies(Pierre,exam)}              A
[3] {-studies(z8,exam), pass(z8,exam)}  A
[4] {-studies(Pierre,exam)}             (1, 3)
[5] {pass(Pierre,exam)}                 (2, 3)
[6] {}                                  (1, 5)

Out[11]: True
```

Thus, you can see from these evaluations that Pierre does pass the exam because he studied for the exam. However, John who doesn't pass the exam since he did not study for it. Let's consider a more complex example with several entities that perform several actions, as follows.

- There are two dogs, Rover (r) and Alex (a)
- There is one cat, Garfield (g)

- There is one fox, Felix (f)
- Two animals—Alex (a) and Felix (f)—run, as denoted by the runs() function
- Two animals—Rover (r) and Garfield (g)—sleep, as denoted by the sleeps() function
- Two animals—Felix (f) and Alex (a)—can jump over the other two, as denoted by the jumps_over() function

Taking all these assumptions, the following snippet builds a FOL-based model with the domain and assignment values based on the entities and functions. Once we build this model, we evaluate various FOL-based expressions to determine their outcomes and prove the theorems, like we did earlier.

```
# define symbols (entities\functions) and their values
rules = """
    rover => r
    felix => f
    garfield => g
    alex => a
    dog => {r, a}
    cat => {g}
    fox => {f}
    runs => {a, f}
    sleeps => {r, g}
    jumps_over => {(f, g), (a, g), (f, r), (a, r)}
    """
val = nltk.Valuation.fromstring(rules)

# view the valuation object of symbols and their assigned values (dictionary)
Val

{'rover': 'r', 'runs': set([('f',), ('a',)]), 'alex': 'a', 'sleeps':
set([('r',), ('g',)]), 'felix': 'f', 'fox': set([('f',)]), 'dog':
set([('a',), ('r',)]), 'jumps_over': set([('a', 'g'), ('f', 'g'), ('a', 'r'),
('f', 'r')]), 'cat': set([('g',)]), 'garfield': 'g'}

# define domain and build FOL based model
dom = {'r', 'f', 'g', 'a'}
```

```
m = nltk.Model(dom, val)

# evaluate various expressions
m.evaluate('jumps_over(felix, rover) & dog(rover) & runs(rover)', None)

False

m.evaluate('jumps_over(felix, rover) & dog(rover) & -runs(rover)', None)

True

m.evaluate('jumps_over(alex, garfield) & dog(alex) & cat(garfield) &
sleeps(garfield)', None)

True

# assign rover to x and felix to y in the domain
g = nltk.Assignment(dom, [('x', 'r'), ('y', 'f')])

# evaluate more expressions based on above assigned symbols
m.evaluate('runs(y) & jumps_over(y, x) & sleeps(x)', g)

True

m.evaluate('exists y. (fox(y) & runs(y))', g)

True
```

This snippet depicts the evaluation of expressions based on the valuation of different symbols, based on the rules and domain. We create FOL-based expressions and see their outcomes based on the predefined assumptions. For example, the first expression returns false because Rover never `runs()` and the second and third expressions are true because they satisfy all the conditions, like Felix and Alex can jump over Rover or Garfield, Rover is a dog, which does not run, and Garfield is a cat.

The second set of expressions is evaluated based on assigning Felix and Rover to specific symbols in our domain (`dom`) and passing that variable (`g`) when evaluating the expressions. We can even satisfy open formulae or expressions using the `satisfiers()` function, as depicted here:

```
# who are the animals who run?
formula = read_expr('runs(x)')
m.satisfiers(formula, 'x', g)
```

```
{'a', 'f'}
```

```
# animals who run and are also a fox?
formula = read_expr('runs(x) & fox(x)')
m.satisfiers(formula, 'x', g)
```

```
{'f'}
```

These outputs are self-explanatory, wherein we evaluate open ended questions like which animals run? Which animals can run and are also foxes? We get the relevant symbols in our outputs, which we can map back to the actual animal names (Hint: `a: alex, f: felix`). We encourage you to experiment with more propositions and FOL expressions by building your own assumptions, domain, and rules.

Summary

In this chapter, we covered a variety of topics focused on semantic analysis of textual data. We revisited several of our concepts from Chapter 1 with regards to language semantics. We looked at the WordNet corpus in detail and explored the concept of synsets with practical examples. We also analyzed various lexical semantic relations from Chapter 1 using synsets and real-world examples. We looked at relationships including entailments, homonyms and homographs, synonyms and antonyms, hyponyms and hypernyms, and holonyms and meronyms.

Semantic relations and similarity computation techniques were also discussed in detail, with examples that leveraged common hypernyms among various synsets. Some popular techniques widely used in semantic and information extraction were also discussed, which included word sense disambiguation and named entity recognition. We looked at state-of-the-art pretrained NER models from spaCy and NLTK, including leveraging Stanford Core NLP NER models. We also learned how to build our own NER tagging model from scratch! Besides semantic relations, we also revisited concepts related to semantic representations, namely propositional logic and first order logic. We leveraged the use of theorem provers and evaluated propositions and logical expressions computationally. The next chapter focuses on one of the most popular applications in NLP, sentiment analysis. Stay tuned!

CHAPTER 9

Sentiment Analysis

In this chapter, we cover one of the most interesting and widely used aspects pertaining to natural language processing (NLP), text analytics, and machine learning. The problem at hand is sentiment analysis or opinion mining, where we want to analyze some textual documents and predict their sentiment or opinion based on the content of these documents. Sentiment analysis is perhaps one of the most popular applications of natural language processing and text analytics, with a vast number of websites, books, and tutorials on this subject. Sentiment analysis seems to work best on subjective text, where people express opinions, feelings, and their mood. From a real-world industry standpoint, sentiment analysis is widely used to analyze corporate surveys, feedback surveys, social media data, and reviews for movies, places, commodities, and many more. The idea is to analyze the reactions of people about a specific entity and take insightful actions based on their sentiments.

A text corpus consists of multiple text documents and each document can be as simple as a single sentence to as complex as a complete document with multiple paragraphs. Textual data, in spite of being highly unstructured, can be classified into two major types of documents. *Factual documents* typically depict some form of statements or facts with no specific feelings or emotion attached to them. These are also known as objective documents. *Subjective documents,* on the other hand, express feelings, mood, emotions, and opinions.

Sentiment analysis is also popularly known as opinion analysis or opinion mining. The key idea is to use techniques from text analytics, NLP, machine learning, and linguistics to extract important information or data points from unstructured text. This in turn can help us derive qualitative outputs like the overall sentiment being on a positive, neutral, or negative scale and quantitative outputs like the sentiment polarity, subjectivity, and objectivity proportions. Sentiment polarity is typically a numeric score assigned to the positive and negative aspects of a text document and is based on

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_9

subjective parameters like specific words and phrases expressing feelings and emotion. *Neutral* sentiments typically have a 0 polarity, since it does not express any specific sentiment, *positive* sentiments have polarity > 0, and *negative* sentiments are < 0. Of course, you can always change these thresholds based on the type of text you are dealing with. There are no hard constraints on this.

In this chapter, we focus on analyzing a large corpus of movie reviews and deriving sentiment from them. We cover a wide variety of techniques for analyzing sentiment, including the following:

- Unsupervised lexicon-based models
- Traditional supervised machine learning models
- Newer supervised deep learning models
- Advanced supervised deep learning models

Besides looking at various approaches and models, we also briefly recap important aspects in the machine learning pipeline around text preprocessing and normalization. Besides this, we also perform an in-depth analysis of our predictive models, including model interpretation and topic models.

The key idea here is to understand how we tackle a problem like sentiment analysis on unstructured text, learn various techniques and models, and understand how to interpret the results. This will enable you to use these methodologies in the future on your own datasets. All the code examples showcased in this chapter are available on the book's official GitHub repository at https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition. Let's get started!

Problem Statement

The main objective in this chapter is to predict the sentiment of a number of movie reviews obtained from the Internet Movie Database (IMDB). This dataset contains 50,000 movie reviews that have been labeled with positive and negative sentiment class labels based on the review content. There are additional movie reviews that are unlabeled. The dataset can be obtained from http://ai.stanford.edu/~amaas/data/sentiment/, courtesy of Stanford University and Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. This dataset was also used in their famous paper, "Learning Word Vectors for Sentiment Analysis," from proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011). They have datasets in the form of raw text as well as already processed Bag of Words formats.

We use the raw labeled movie reviews for our analyses in this chapter. Hence our task is to predict the sentiment of 15,000 labeled movie reviews and use the remaining 35,000 reviews to train our supervised models. We will still predict sentiments for 15,000 reviews in the case of unsupervised models to maintain consistency and enable ease of comparison.

Setting Up Dependencies

We will be using several Python libraries and frameworks specific to text analytics, NLP, and machine learning. While most of them are mentioned in each section, you need to make sure you have Pandas, NumPy, SciPy, and Scikit-Learn installed, which are used for data processing and machine learning. Deep learning frameworks used in this chapter include Keras with the TensorFlow backend. NLP libraries we use include spaCy, NLTK, and Gensim. We also use our custom developed text preprocessing and normalization module from Chapter 3, which you can find in the files named `contractions.py` and `text_normalizer.py`. Utilities related to supervised model fitting, prediction, and evaluation are present in `model_evaluation_utils.py`, so make sure you place these modules in the same directory and the other Python files and Jupyter notebooks for this chapter.

Getting the Data

The dataset will be available along with the code files for this chapter in the GitHub repository for this book at `https://github.com/dipanjanS/text-analytics-with-python` under the file called `movie_reviews.csv`. It contains 50,000 labeled IMDB movie reviews. This should be present in the corresponding notebooks folder for this chapter under the directory for the Second Edition of the book. You can also download the data from `http://ai.stanford.edu/~amaas/data/sentiment/` if needed. Once you have the CSV file, you can easily load it in Python using the `read_csv(...)` utility function from Pandas.

Text Preprocessing and Normalization

One of the key steps before diving into the process of feature engineering and modeling involves cleaning, preprocessing, and normalizing text to bring text components like phrases and words to a standard format. We talked about this several times because it is one of the most crucial stages for any NLP pipeline. Preprocessing enables standardization across a document corpus, which helps build meaningful features and reduce noise that can be introduced due to many factors, such as irrelevant symbols, special characters, XML and HTML tags, and so on. Our `text_normalizer` module built in Chapter 3 contains all the necessary utilities for our text normalization needs. You can also refer to a sample Jupyter notebook named `Text Normalization Demo.ipynb` for a more interactive experience. Just to refresh your memory, the main components in our text normalization pipeline are described in this section.

- **Cleaning text:** Our text often contains unnecessary content like HTML tags, which do not add much value when analyzing sentiment. Hence, we need to make sure we remove them before extracting features. The BeautifulSoup library does an excellent job in providing necessary functions for this. Our `strip_html_tags(...)` function cleans and strips out HTML code.
- **Removing accented characters:** In our dataset, we are dealing with reviews in the English language so we need to make sure that accented characters are converted and standardized into ASCII characters. A simple example is converting **é** to **e**. Our `remove_accented_chars(...)` function helps us in this respect.
- **Expanding contractions:** In the English language, contractions are shortened versions of words. These shortened versions of existing words or phrases are created by removing specific letters and sounds. More often than not, vowels are removed from the words. Examples include do not to don't and I would to I'd. Contractions pose a problem in text normalization because we have to deal with special characters like the apostrophe and we also have to convert each contraction to its expanded, original form. Our `expand_contractions(...)` function uses regular expressions and various contractions mapped in our `contractions.py` module to expand all contractions in our text corpus.

- **Removing special characters:** Another important task in text cleaning and normalization is to remove special characters and symbols that add to the noise in unstructured text. Simple regexes can be used to achieve this. Our `remove_special_characters(...)` function removes special characters. In our code, we have retained numbers but you can also remove numbers if you do not want them in your normalized corpus.

- **Stemming and lemmatization:** Word stems are usually the base form of possible words, which can be created by attaching affixes, like prefixes and suffixes, to the stem to create new words. This is known as *inflection*. The reverse process of obtaining the base form of a word is known as stemming. A simple example is ***WATCHES***, ***WATCHING***, and ***WATCHED***, which have the word root stem ***WATCH***. The NLTK package offers a wide range of stemmers, like the `PorterStemmer` and `LancasterStemmer`. Lemmatization is very similar to stemming, where we remove word affixes to get to the base form of a word. However, the base form is known as the root word not the root stem. The difference being that the root word is always a lexicographically correct word (present in the dictionary) but the root stem may not correct. We use lemmatization only in our normalization pipeline to retain lexicographically correct words. The `lemmatize_text(...)` function helps us in this regard.

- **Removing stopwords:** Words that have little or no significance, especially when constructing meaningful features from text, are known as stopwords. These are usually words that end up having the maximum frequency if you do a simple term or word frequency in a document corpus. Words like “a,” “an,” “the,” and so on are stopwords. There is no universal stopword list, but we use a standard English language stopwords list from NLTK. You can also add your own domain specific stopwords if needed. The `remove_stopwords(...)` function removes stopwords and retains words having the most significance and context in a corpus.

We use these components and tie them together in the following function called `normalize_corpus(...)`, which can be used to take a document corpus as input and return the same corpus with cleaned and normalized text documents. Refer to Chapter 3 to do a more detailed recap around text preprocessing. Now that we have our normalization module ready, we can start modeling and analyzing our corpus.

Unsupervised Lexicon-Based Models

We talked about unsupervised learning methods in the past, which refer to specific modeling methods that can be applied directly to data features without the presence of labeled data. One of the major challenges in any organization is getting labeled datasets due the lack of time as well as resources to do this tedious task. Unsupervised methods are very useful in this scenario and we look at some of these methods in this section. Even though we have labeled data, this section should give you a good idea of how lexicon based models work and you can apply them to your own datasets when you do not have labeled data.

Unsupervised sentiment analysis models use well curated knowledgebases, ontologies, lexicons, and databases, which have detailed information pertaining to subjective words, phrases including sentiment, mood, polarity, objectivity, subjectivity, and so on. A lexicon model typically uses a *lexicon*, also known as a dictionary or vocabulary of words specifically aligned to sentiment analysis. These lexicons contain a list of words associated with positive and negative sentiment, polarity (magnitude of negative or positive score), parts of speech (POS) tags, subjectivity classifiers (strong, weak, neutral), mood, modality, and so on. You can use these lexicons and compute the sentiment of a text document by matching the presence of specific words from the lexicon and then looking at other factors like presence of negation parameters, surrounding words, overall context, phrases, and aggregate overall sentiment polarity scores to decide the final sentiment score. There are several popular lexicon models used for sentiment analysis. Some of them are as follows:

- Bing Liu's lexicon
- MPQA subjectivity lexicon
- Pattern lexicon
- TextBlob lexicon
- AFINN lexicon

- SentiWordNet lexicon
- VADER lexicon

This is not an exhaustive list of lexicon models but these are definitely among the most popular ones available today. We cover the last three lexicon models in more detail with hands-on code and examples using our movie review dataset. We use the last 15,000 reviews and predict their sentiment to see how well our model performs based on model evaluation metrics like accuracy, precision, recall, and F1-score (which we covered in detail in Chapter 5). Since we have labeled data, it will be easy for us to see how well our sentiment values for these movie reviews match our lexicon-model based predicted sentiment values. You can refer to the Jupyter notebook titled `Sentiment Analysis - Unsupervised Lexical.ipynb` for an interactive experience. Before we start our analysis, let's load the necessary dependencies and configuration settings using the following snippet.

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import text_normalizer as tn
   ...: import model_evaluation_utils as meu
   ...:
   ...: np.set_printoptions(precision=2, linewidth=80)
```

Now, we can load our IMDB review dataset and subset out the last 15,000 reviews for our analysis. We don't need to normalize them since most of the frameworks we will be using handle this internally, but for some we might use some basic preprocessing steps as needed.

```
In [2]: dataset = pd.read_csv('movie_reviews.csv.bz2',
                              compression='bz2')
   ...:
   ...: reviews = np.array(dataset['review'])
   ...: sentiments = np.array(dataset['sentiment'])
   ...:
   ...: # extract data for model evaluation
   ...: test_reviews = reviews[35000:]
   ...: test_sentiments = sentiments[35000:]
   ...: sample_review_ids = [7626, 3533, 13010]
```

We also extract some sample reviews so that we can run our models on them and interpret their results in detail.

Bing Liu's Lexicon

This lexicon contains over 6,800 words, which have been divided into two files named positive-words.txt, containing around 2,000 words/phrases, and negative-words.txt, which contains over 4,800 words/phrases. The lexicon has been developed and curated by Bing Liu over several years and has also been explained in detail in his original paper by Nitin Jindal and Bing Liu, entitled "Identifying Comparative Sentences in Text Documents," from the proceedings of the 29th Annual International ACM SIGIR, Seattle 2006. If you want to use this lexicon, you can get it from https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon, which includes a link to download it as an archive (RAR format).

MPQA Subjectivity Lexicon

The term MPQA stands for Multi-Perspective Question Answering and it contains a diverse set of resources pertaining to opinion corpora, subjectivity lexicon, subjectivity sense annotations, argument lexicon, debate corpora, opinion finder, and many more. This is developed and maintained by the University of Pittsburgh and their official website at http://mpqa.cs.pitt.edu/ contains all the necessary information. The subjectivity lexicon is a part of their opinion finder framework and contains subjectivity clues and contextual polarity. Details about this can be found in the paper by Theresa Wilson, Janyce Wiebe, and Paul Hoffmann, entitled "Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis" from the proceedings of HLT-EMNLP-2005. You can download the subjectivity lexicon from their official website at http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/. It contains subjectivity clues present in the dataset named subjclueslen1-HLTEMNLP05.tff. The following snippet shows some sample lines from the lexicon.

```
type=weaksubj len=1 word1=abandonment pos1=noun stemmed1=n
priorpolarity=negative
type=weaksubj len=1 word1=abandon pos1=verb stemmed1=y
priorpolarity=negative
...
...
type=strongsubj len=1 word1=zenith pos1=noun stemmed1=n
priorpolarity=positive
type=strongsubj len=1 word1=zest pos1=noun stemmed1=n
priorpolarity=positive
```

Each line consists of a specific word and its associated polarity, POS tag information, length (right now only words of length 1 are present), subjective context, and stem information.

Pattern Lexicon

The pattern package is a complete natural language processing framework available in Python and can be used for text processing, sentiment analysis, and more. This has been developed by CLiPS (Computational Linguistics and Psycholinguistics), a research center associated with the Linguistics Department of the Faculty of Arts of the University of Antwerp. The pattern uses its own sentiment module, which internally uses a lexicon that you can access from their official GitHub repository at `https://github.com/clips/pattern/blob/master/pattern/text/en/en-sentiment.xml`. It this contains the complete subjectivity-based lexicon database. Each line in the lexicon typically looks like the following sample.

```
<word form="absurd" wordnet_id="a-02570643" pos="JJ" sense="incongruous"
polarity="-0.5" subjectivity="1.0" intensity="1.0" confidence="0.9" />
```

Thus you get important metadata information like WordNet corpus identifiers, polarity scores, word sense, POS tags, intensity, subjectivity scores, and so on. These can in turn be used to compute sentiment over a text document based on polarity and subjectivity. Unfortunately, the pattern has still not been ported officially for Python 3.x and it works on Python 2.7.x. However, you can load up this lexicon and do your own modeling as needed. Even better, the popular framework TextBlob uses this for sentiment analysis and is available in Python 3!

TextBlob Lexicon

As mentioned, the pattern package has a nice module for sentiment analysis but is sadly only available for Python 2.7.x. However, our focus is on building applications on Python 3.x, so we can use TextBlob out of the box! The lexicon that TextBlob uses is the same one as pattern and is available in their source code on GitHub (`https://github.com/sloria/TextBlob/blob/dev/textblob/en/en-sentiment.xml`). Some sample examples are shown from the lexicon as follows.

```
<word form="abhorrent" wordnet_id="a-1625063" pos="JJ" sense="offensive
to the mind" polarity="-0.7" subjectivity="0.8" intensity="1.0"
reliability="0.9" />
<word form="able" cornetto_synset_id="n_a-534450" wordnet_id="a-01017439"
pos="JJ" sense="having a strong healthy body" polarity="0.5"
subjectivity="1.0" intensity="1.0" confidence="0.9" />
```

Typically, specific adjectives have a polarity score (negative/positive, -1.0 to +1.0) and a subjectivity score (objective/subjective, +0.0 to +1.0). The reliability score specifies if an adjective was hand-tagged (1.0) or inferred (0.7). Words are tagged per sense, e.g., ridiculous (pitiful) = negative, ridiculous (humorous) = positive. The Cornetto id (lexical unit id) and Cornetto synset id refer to the Cornetto lexical database for Dutch. The WordNet id refers to the WordNet3 lexical database for English. The part-of-speech tags (POS) use the Penn Treebank convention. Let’s look at how we can use TextBlob for sentiment analysis.

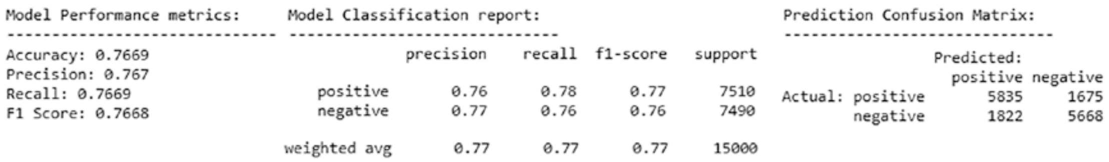
```
for review, sentiment in zip(test_reviews[sample_review_ids], test_
sentiments[sample_review_ids]):
    print('REVIEW:', review)
    print('Actual Sentiment:', sentiment)
    print('Predicted Sentiment polarity:', textblob.TextBlob(review).
    sentiment.polarity)
    print('-'*60)
```

```
REVIEW: no comment - stupid movie, acting average or worse... screenplay -
no sense at all... SKIP IT!
Actual Sentiment: negative
Predicted Sentiment polarity: -0.3625
------------------------------------------------------------
REVIEW: I don't care if some people voted this movie to be bad. If you want
the Truth this is a Very Good Movie! It has every thing a movie should
have. You really should Get this one.
Actual Sentiment: positive
Predicted Sentiment polarity: 0.16666666666666674
------------------------------------------------------------
```

```
REVIEW: Worst horror film ever but funniest film ever rolled in one you
have got to see this film it is so cheap it is unbelievable but you have to
see it really!!!! P.s watch the carrot
Actual Sentiment: positive
Predicted Sentiment polarity: -0.037239583333333326
------------------------------------------------------------
```

You can check the sentiment of some specific movie reviews and the sentiment polarity score as predicted by TextBlob. Typically, a positive score denotes positive sentiment and a negative score denotes negative sentiment. You can use a specific custom threshold to determine what should be positive or negative. We use a custom threshold of 0.1 based on multiple experiments. The following code computes the sentiment on the entire test data. See Figure 9-1.

```
sentiment_polarity = [textblob.TextBlob(review).sentiment.polarity for
                      review in test_reviews]
predicted_sentiments = ['positive' if score >= 0.1 else 'negative'
                                   for score in sentiment_polarity]
meu.display_model_performance_metrics(true_labels=test_sentiments,
                                      predicted_labels=predicted_sentiments,
                                      classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                              Prediction Confusion Matrix:
------------------------------ ------------------------------                            ------------------------------
Accuracy: 0.7669                            precision    recall  f1-score   support                    Predicted:
Precision: 0.767                                                                                        positive negative
Recall: 0.7669                    positive       0.76      0.78      0.77      7510     Actual: positive     5835     1675
F1 Score: 0.7668                  negative       0.77      0.76      0.76      7490             negative     1822     5668

                               weighted avg       0.77      0.77      0.77     15000
```

Figure 9-1. *Model performance metrics for pattern lexicon based model*

We get an overall F1-score and accuracy of 77%, which is good considering it's an unsupervised model! Looking at the confusion matrix, we can clearly see that we have an equal number of reviews almost being misclassified as positive and negative, which gives consistent results with regard to precision and recall for each class.

AFINN Lexicon

The AFINN lexicon is perhaps one of the simplest and most popular lexicons and can be used extensively for sentiment analysis. Developed and curated by Finn Årup Nielsen, you can find more details on this lexicon in the paper by Finn Årup Nielsen, entitled "A New ANEW: Evaluation of a Word List for Sentiment Analysis in Microblogs," from the proceedings of the ESWC2011 workshop. The current version of the lexicon is `AFINN-en-165.txt` and it contains over 3,300 words with a polarity score associated with each word.

You can find this lexicon at the author's official GitHub repository along with previous versions of this lexicon including `AFINN-111` at https://github.com/fnielsen/afinn/blob/master/afinn/data/. The author has also created a nice wrapper library on top of this in Python called `afinn`, which we will be using for our analysis needs. You can import the library and instantiate an object using the following code.

```
In [3]: from afinn import Afinn
   ...:
   ...: afn = Afinn(emoticons=True)
```

We can now use this object and compute the polarity of our chosen four sample reviews using the following snippet.

```
In [4]: for review, sentiment in zip(test_reviews[sample_review_ids], test_
sentiments[sample_review_ids]):
   ...:     print('REVIEW:', review)
   ...:     print('Actual Sentiment:', sentiment)
   ...:     print('Predicted Sentiment polarity:', afn.score(review))
   ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay -
no sense at all... SKIP IT!
Actual Sentiment: negative
Predicted Sentiment polarity: -7.0
------------------------------------------------------------
REVIEW: I don't care if some people voted this movie to be bad. If you want
the Truth this is a Very Good Movie! It has every thing a movie should
have. You really should Get this one.
Actual Sentiment: positive
Predicted Sentiment polarity: 3.0
```

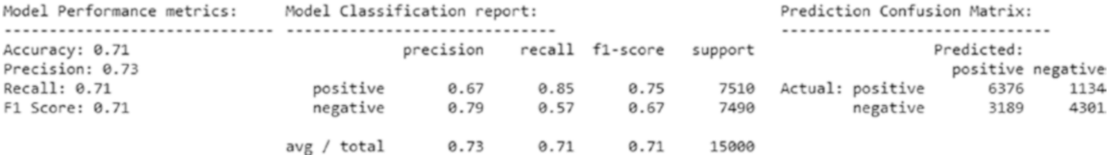
```
------------------------------------------------------------
REVIEW: Worst horror film ever but funniest film ever rolled in one you
have got to see this film it is so cheap it is unbelievable but you have to
see it really!!!! P.s watch the carrot
Actual Sentiment: positive
Predicted Sentiment polarity: -3.0
------------------------------------------------------------
```

We can compare the actual sentiment label for each review and check out the predicted sentiment polarity score. A negative polarity typically denotes negative sentiment. To predict sentiment on our complete test dataset of 15,000 reviews (I used the raw text documents because AFINN takes into account other aspects like emoticons and exclamations), we can now use the following snippet. I used a threshold of >= 1.0 to determine if the overall sentiment is positive. You can choose your own threshold based on analyzing your own corpora.

```
In [5]: sentiment_polarity = [afn.score(review) for review in test_reviews]
   ...: predicted_sentiments = ['positive' if score >= 1.0 else 'negative'
   for score in sentiment_polarity]
```

Now that we have our predicted sentiment labels, we can evaluate our model performance based on standard performance metrics using our utility function. See Figure 9-2.

```
In [6]: meu.display_model_performance_metrics(true_labels=test_sentiments,
predicted_labels=predicted_sentiments, classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                       Prediction Confusion Matrix:
------------------------------ ------------------------------                     ------------------------------
Accuracy: 0.71                              precision    recall  f1-score   support                  Predicted:
Precision: 0.73                                                                                    positive negative
Recall: 0.71                       positive       0.67      0.85      0.75      7510  Actual: positive     6376     1134
F1 Score: 0.71                     negative       0.79      0.57      0.67      7490          negative     3189     4301

                               avg / total       0.73      0.71      0.71     15000
```

Figure 9-2. *Model performance metrics for AFINN lexicon based model*

We get an overall F1-score of 71%, which is quite decent considering it's an unsupervised model. Looking at the confusion matrix, we can clearly see that quite a number of *negative* sentiment-based reviews have been misclassified as positive (3,189) and this leads to the lower recall of 57% for the negative sentiment class. Performance for the positive class is better with regard to recall or hit-rate, where we correctly predicted 6,376 out of 7,510 positive reviews, but the precision is 67% because of the many wrong positive predictions made in case of the negative sentiment reviews.

SentiWordNet Lexicon

The WordNet corpus is one of the most popular corpora for the English language and is used extensively in natural language processing and semantic analysis. WordNet gave us the concept of synsets or synonym sets. The SentiWordNet lexicon is based on WordNet synsets and can be used for sentiment analysis and opinion mining. The SentiWordNet lexicon typically assigns three sentiment scores for each WordNet synset. These include a positive polarity score, a negative polarity score, and an objectivity score. Further details are available on the official website at http://sentiwordnet.isti.cnr.it, including research papers and download links for the lexicon. We use the NLTK library, which provides a Pythonic interface into SentiWordNet. Consider we have the adjective "awesome". We can get the sentiment scores associated with the synset for this word using the following snippet.

```
In [8]: from nltk.corpus import sentiwordnet as swn
   ...:
   ...: awesome = list(swn.senti_synsets('awesome', 'a'))[0]
   ...: print('Positive Polarity Score:', awesome.pos_score())
   ...: print('Negative Polarity Score:', awesome.neg_score())
   ...: print('Objective Score:', awesome.obj_score())
Positive Polarity Score: 0.875
Negative Polarity Score: 0.125
Objective Score: 0.0
```

Let's now build a generic function to extract and aggregate sentiment scores for a complete textual document based on matched synsets in that document.

```
def analyze_sentiment_sentiwordnet_lexicon(review, verbose=False):

    # tokenize and POS tag text tokens
    tagged_text = [(token.text, token.tag_) for token in tn.nlp(review)]
    pos_score = neg_score = token_count = obj_score = 0
    # get wordnet synsets based on POS tags
    # get sentiment scores if synsets are found
    for word, tag in tagged_text:
        ss_set = None
        if 'NN' in tag and list(swn.senti_synsets(word, 'n')):
            ss_set = list(swn.senti_synsets(word, 'n'))[0]
        elif 'VB' in tag and list(swn.senti_synsets(word, 'v')):
            ss_set = list(swn.senti_synsets(word, 'v'))[0]
        elif 'JJ' in tag and list(swn.senti_synsets(word, 'a')):
            ss_set = list(swn.senti_synsets(word, 'a'))[0]
        elif 'RB' in tag and list(swn.senti_synsets(word, 'r')):
            ss_set = list(swn.senti_synsets(word, 'r'))[0]
        # if senti-synset is found
        if ss_set:
            # add scores for all found synsets
            pos_score += ss_set.pos_score()
            neg_score += ss_set.neg_score()
            obj_score += ss_set.obj_score()
            token_count += 1

    # aggregate final scores
    final_score = pos_score - neg_score
    norm_final_score = round(float(final_score) / token_count, 2)
    final_sentiment = 'positive' if norm_final_score >= 0 else 'negative'
    if verbose:
        norm_obj_score = round(float(obj_score) / token_count, 2)
        norm_pos_score = round(float(pos_score) / token_count, 2)
        norm_neg_score = round(float(neg_score) / token_count, 2)
        # to display results in a nice table
        sentiment_frame = pd.DataFrame([[final_sentiment, norm_obj_score,
                                        norm_pos_score, norm_neg_score,
                                        norm_final_score]],
                                      columns=pd.MultiIndex (levels=[
                                      ['SENTIMENT STATS:'],
                                      ['Predicted Sentiment', 'Objectivity',
                                       'Positive', 'Negative', 'Overall']],
                                      labels=[[0,0,0,0,0],[0,1,2,3,4]]))
        print(sentiment_frame)

    return final_sentiment
```

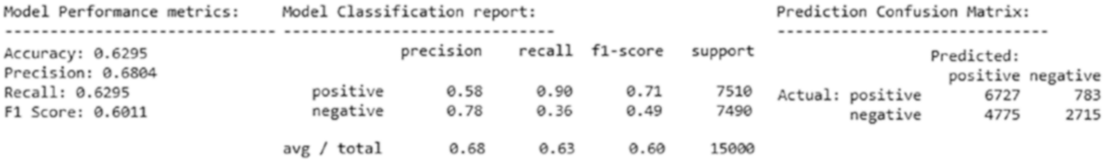
Our function takes in a movie review, tags each word with its corresponding POS tag, extracts the sentiment scores for any matched synset token based on its POS tag, and finally aggregates the scores. This process will be clearer when we run it on our sample documents.

```
In [10]: for review, sentiment in zip(test_reviews[sample_review_ids],
test_sentiments[sample_review_ids]):
    ...:     print('REVIEW:', review)
    ...:     print('Actual Sentiment:', sentiment)
    ...:     pred = analyze_sentiment_sentiwordnet_lexicon(review,
             verbose=True)
    ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay -
no sense at all... SKIP IT!
Actual Sentiment: negative
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0             negative        0.76     0.09     0.15   -0.06
------------------------------------------------------------
REVIEW: I don't care if some people voted this movie to be bad. If you want
the Truth this is a Very Good Movie! It has every thing a movie should
have. You really should Get this one.
Actual Sentiment: positive
     SENTIMENT STATS:
```

```
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive        0.76      0.19     0.06    0.13
------------------------------------------------------------
REVIEW: Worst horror film ever but funniest film ever rolled in one you
have got to see this film it is so cheap it is unbelievable but you have to
see it really!!!! P.s watch the carrot
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            positive         0.8     0.12     0.07    0.05
------------------------------------------------------------
```

We can clearly see the predicted sentiment along with sentiment polarity scores and an objectivity score for each sample movie review depicted in formatted dataframes. Let's use this model to predict the sentiment of all our test reviews and evaluate its performance. A threshold of >=0 has been used for the overall sentiment polarity to be classified as positive (whereas < 0 is a negative sentiment). See Figure 9-3.

```
In [11]:  norm_test_reviews = tn.normalize_corpus(test_reviews)
    ...: predicted_sentiments = [analyze_sentiment_sentiwordnet_
         lexicon(review, verbose=False) for review in norm_test_reviews]
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
         predicted_labels=predicted_sentiments,
    ...: classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                        Prediction Confusion Matrix:
------------------------------ ------------------------------                      ------------------------------
Accuracy: 0.6295                            precision    recall  f1-score   support
Precision: 0.6804                                                                                 Predicted:
Recall: 0.6295                     positive       0.58      0.90      0.71      7510                 positive negative
F1 Score: 0.6011                   negative       0.78      0.36      0.49      7490  Actual: positive     6727      783
                                                                                              negative     4775     2715
                                avg / total       0.68      0.63      0.60     15000
```

Figure 9-3. *Model performance metrics for SentiWordNet lexicon based model*

We get an overall F1-score of 60%, which is definitely a step down from our previous models. We can see a large number of negative reviews being misclassified as positive. Maybe playing around with the thresholds here might help!

VADER Lexicon

The VADER lexicon, developed by C.J. Hutto, is based on a rule-based sentiment analysis framework, specifically tuned to analyze sentiments in social media. VADER stands for Valence Aware Dictionary and sEntiment Reasoner. Details about this framework can be read in the original paper by Hutto, C.J., and Gilbert, E.E. (2014), entitled "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text," from the proceedings of the Eighth International Conference on Weblogs and Social Media (ICWSM-14). You can use the library based on NLTK's interface under the `nltk.sentiment.vader` module.

You can also download the actual lexicon or install the framework from https://github.com/cjhutto/vaderSentiment, which also contains detailed information about VADER. This lexicon, present in the file titled `vader_lexicon.txt`, contains necessary sentiment scores associated with words, emoticons, and slangs (like wtf, lol, nah, and so on). There were a total of over 9,000 lexical features from which over 7,500 curated lexical features were finally selected in the lexicon with proper validated valence scores. Each feature was rated on a scale from `"[-4] Extremely Negative"` to `"[4] Extremely Positive"`, with allowance for `"[0] Neutral (or Neither, N/A)"`.

The process of selecting lexical features was done by keeping all features that had a non-zero mean rating and whose standard deviation was less than 2.5, which was determined by the aggregate of ten independent raters. We depict a sample from the VADER lexicon as follows:

```
:(    -1.9   1.13578     [-2, -3, -2, 0, -1, -1, -2, -3, -1, -4]
:)    2.0    1.18322     [2, 2, 1, 1, 1, 1, 4, 3, 4, 1]
...
terrorizing  -3.0  1.0         [-3, -1, -4, -4, -4, -3, -2, -3, -2, -4]
thankful     2.7   0.78102     [4, 2, 2, 3, 2, 4, 3, 3, 2, 2]
```

Each line in the preceding lexicon sample depicts a unique term, which can either be an emoticon or a word. The first token indicates the word/emoticon, the second token indicates the mean sentiment polarity score, the third token indicates the standard deviation, and the final token indicates a list of scores given by 10 independent scorers. Now let's use VADER to analyze our movie reviews! We build our own modeling function as follows.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

def analyze_sentiment_vader_lexicon(review,
                                    threshold=0.1,
                                    verbose=False):
    # preprocess text
    review = tn.strip_html_tags(review)
    review = tn.remove_accented_chars(review)
    review = tn.expand_contractions(review)

    # analyze the sentiment for review
    analyzer = SentimentIntensityAnalyzer()
    scores = analyzer.polarity_scores(review)
    # get aggregate scores and final sentiment
    agg_score = scores['compound']
    final_sentiment = 'positive' if agg_score >= threshold\
                                   else 'negative'
    if verbose:
        # display detailed sentiment statistics
        positive = str(round(scores['pos'], 2)*100)+'%'
        final = round(agg_score, 2)
        negative = str(round(scores['neg'], 2)*100)+'%'
        neutral = str(round(scores['neu'], 2)*100)+'%'
        sentiment_frame = pd.DataFrame([[final_sentiment, final, positive,
                                        negative, neutral]],
                                        columns=pd.MultiIndex(levels=
                                                      [['SENTIMENT STATS:'],
                                        ['Predicted Sentiment', 'Polarity Score',
                                         'Positive', 'Negative', 'Neutral']],
                                        labels=[[0,0,0,0,0],[0,1,2,3,4]]))
        print(sentiment_frame)

    return final_sentiment
```

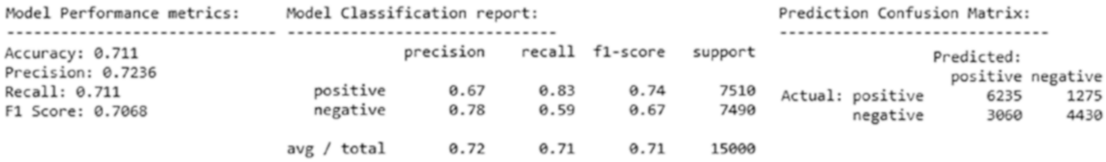
In our modeling function, we do some basic preprocessing but keep the punctuations and emoticons intact. Besides this, we use VADER to get the sentiment polarity and proportion of the review text with regard to positive, neutral, and negative sentiment. We also predict the final sentiment based on a user-input threshold for the aggregated sentiment polarity. Typically, VADER recommends using positive sentiment for aggregated polarity >= 0.5, neutral between [-0.5, 0.5], and negative for polarity < -0.5. We use a threshold of >= 0.4 for positive and < 0.4 for negative in our corpus. The following is the analysis on our sample reviews.

```
In [13]: for review, sentiment in zip(test_reviews[sample_review_ids],
test_sentiments[sample_review_ids]):
    ...:     print('REVIEW:', review)
    ...:     print('Actual Sentiment:', sentiment)
    ...:     pred = analyze_sentiment_vader_lexicon(review, threshold=0.4,
             verbose=True)
    ...:     print('-'*60)
REVIEW: no comment - stupid movie, acting average or worse... screenplay -
no sense at all... SKIP IT!
Actual Sentiment: negative
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           negative           -0.8     0.0%    40.0%   60.0%
------------------------------------------------------------
REVIEW: I don't care if some people voted this movie to be bad. If you want
the Truth this is a Very Good Movie! It has every thing a movie should
have. You really should Get this one.
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive            Negative Neutral
0           negative          -0.16    16.0%  14.0%   69.0%
------------------------------------------------------------
REVIEW: Worst horror film ever but funniest film ever rolled in one you
have got to see this film it is so cheap it is unbelievable but you have to
see it really!!!! P.s watch the carrot
```

```
Actual Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0            positive          0.49    11.0%    11.0%   77.0%
------------------------------------------------------------
```

We can see the detailed statistics pertaining to the sentiment and polarity for each sample movie review. Let's try our model on the complete test movie review corpus and evaluate the model performance.

```
In [14]: predicted_sentiments = [analyze_sentiment_vader_lexicon(review,
threshold=0.4, verbose=False) for review in test_reviews]
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
         predicted_labels=predicted_sentiments,
    ...: classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                         Prediction Confusion Matrix:
------------------------------ ------------------------------                       ------------------------------
Accuracy: 0.711                             precision    recall  f1-score   support
Precision: 0.7236                                                                                  Predicted:
Recall: 0.711                      positive       0.67      0.83      0.74      7510                positive negative
F1 Score: 0.7068                   negative       0.78      0.59      0.67      7490   Actual: positive      6235     1275
                                                                                              negative      3060     4430
                               avg / total       0.72      0.71      0.71     15000
```

__Figure 9-4.__ Model performance metrics for VADER lexicon based model

Figure 9-4 shows an overall F1-score and model accuracy of 71%, which is quite similar to the AFINN-based model. The AFINN-based model wins out on the average precision by only 1%; otherwise, both models have a similar performance.

Classifying Sentiment with Supervised Learning

Another way to build a model to understand the text content and predict the sentiment of the text-based reviews is to use supervised machine learning. To be more specific, we use classification models for solving this problem. We covered the concepts relevant to supervised learning and classification in Chapter 1 under the section "Supervised Learning". With regard to details on building and evaluating classification models, you can head over to Chapter 5 and refresh your memory if needed. We build an automated sentiment text classification system in subsequent sections. The major steps to achieve this are as follows:

1. Prepare train and test datasets (optionally a validation dataset).
2. Preprocess and normalize text documents.
3. Feature engineering.
4. Model training.
5. Model prediction and evaluation.

These are the major steps for building our system. The last optional step is to deploy the model in your server or on the cloud. Figure 9-5 shows a detailed workflow for building a standard text classification system with supervised learning (classification) models.

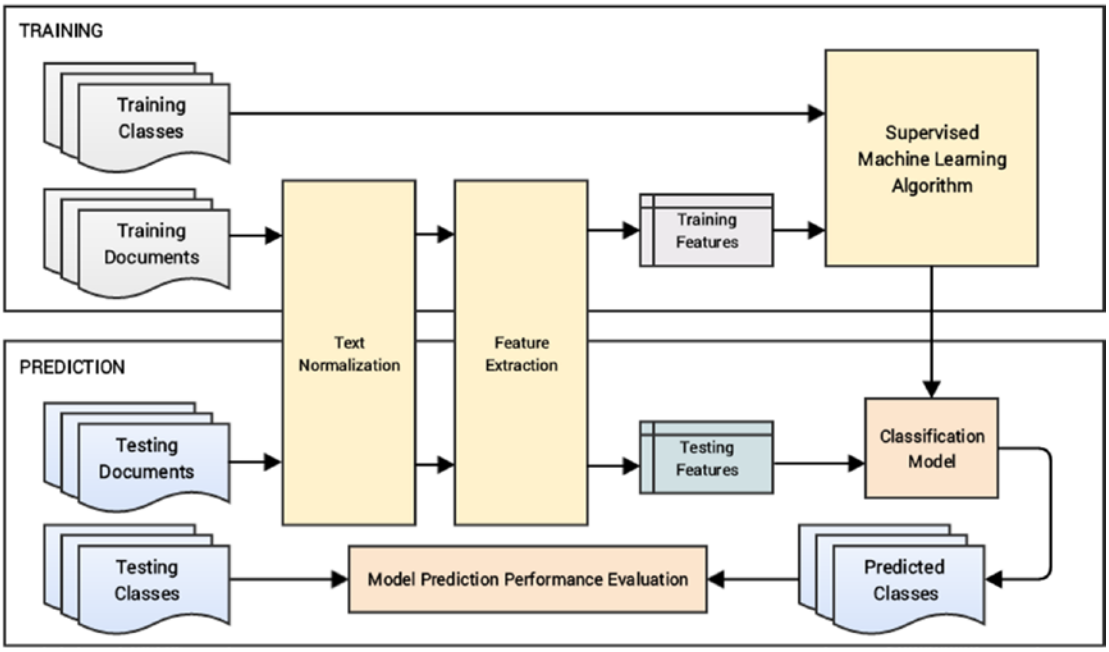


Figure 9-5. *Blueprint for building an automated text classification system*

In our scenario, documents indicate the movie reviews and classes indicate the review sentiments, which can either be positive or negative, making it a binary classification problem. We will build models using traditional machine learning methods and the newer deep learning in the subsequent sections. You can refer to the the Jupyter notebook titled `Sentiment Analysis - Supervised.ipynb` for an interactive experience. Let's load the necessary dependencies and settings before getting started.

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import text_normalizer as tn
   ...: import model_evaluation_utils as meu
   ...: import nltk
   ...: np.set_printoptions(precision=2, linewidth=80)
```

We can now load our IMDB movie reviews dataset, use the first 35,000 reviews for training models, and save the remaining 15,000 reviews as the test dataset to evaluate model performance. Besides this, we also use our normalization module to normalize our review datasets (Steps 1 and 2 in our workflow).

```
In [2]: dataset = pd.read_csv('movie_reviews.csv.bz2',
                              compression='bz2')
   ...:
   ...: # take a peek at the data
   ...: print(dataset.head())
   ...: reviews = np.array(dataset['review'])
   ...: sentiments = np.array(dataset['sentiment'])
   ...:
   ...: # build train and test datasets
   ...: train_reviews = reviews[:35000]
   ...: train_sentiments = sentiments[:35000]
   ...: test_reviews = reviews[35000:]
   ...: test_sentiments = sentiments[35000:]
   ...:
   ...: # normalize datasets
   ...: stop_words = nltk.corpus.stopwords.words('english')
   ...: stop_words.remove('no')
   ...: stop_words.remove('but')
   ...: stop_words.remove('not')
   ...:
```

```
   ...: norm_train_reviews = tn.normalize_corpus(train_reviews)
   ...: norm_test_reviews = tn.normalize_corpus(test_reviews)
                                                 review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive
```

Our datasets are now prepared and normalized so we can proceed from Step 3 in our text classification workflow to build our classification system.

Traditional Supervised Machine Learning Models

We use traditional classification models in this section to classify the sentiment of our movie reviews. Our feature engineering techniques (Step 3) will be based on the Bag of Words model and the TF-IDF model, which were discussed extensively in the section titled "Feature Engineering on Text Data" in Chapter 4. The following snippet helps us engineer features using both these models on our train and test datasets.

```
In [3]: from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
   ...:
   ...: # build BOW features on train reviews
   ...: cv = CountVectorizer(binary=False, min_df=0.0, max_df=1.0, ngram_
        range=(1,2))
   ...: cv_train_features = cv.fit_transform(norm_train_reviews)
   ...: # build TFIDF features on train reviews
   ...: tv = TfidfVectorizer(use_idf=True, min_df=0.0, max_df=1.0, ngram_
        range=(1,2), sublinear_tf=True)
   ...: tv_train_features = tv.fit_transform(norm_train_reviews)
   ...:
   ...: # transform test reviews into features
   ...: cv_test_features = cv.transform(norm_test_reviews)
   ...: tv_test_features = tv.transform(norm_test_reviews)
   ...:
```

```
   ...: print('BOW model:> Train features shape:', cv_train_features.shape,
                ' Test features shape:', cv_test_features.shape)
   ...: print('TFIDF model:> Train features shape:', tv_train_features.
        shape, ' Test features shape:', tv_test_features.shape)

BOW model:> Train features shape: (35000, 2090724)  Test features shape:
(15000, 2090724)
TFIDF model:> Train features shape: (35000, 2090724)  Test features shape:
(15000, 2090724)
```

We take into account word as well as bi-grams for our feature sets. We can now use some traditional supervised machine learning algorithms, which work very well on text classification. We recommend using logistic regression, support vector machines, and multinomial Naïve Bayes models when you work on your own datasets in the future. In this chapter, we built models using logistic regression as well as SVM. The following snippet helps in initializing these classification model estimators.

```
In [4]: from sklearn.linear_model import SGDClassifier, LogisticRegression
   ...:
   ...: lr = LogisticRegression(penalty='l2', max_iter=100, C=1)
   ...: svm = SGDClassifier(loss='hinge', max_iter=100)
```

Without going into too many theoretical complexities, the logistic regression model is a supervised linear machine learning model used for classification regardless of its name. In this model, we try to predict the probability that a given movie review will belong to one of the discrete classes (binary classes in our scenario). The function used by the model for learning is represented here:

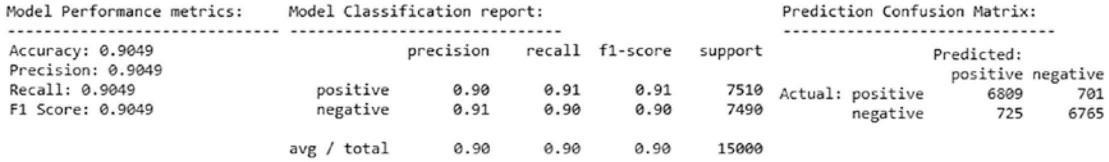
$$P\left(y = positive \mid X\right) = \sigma\left(\theta^T X\right)$$

$$P(y = negative \mid X) = 1 - \sigma\left(\theta^T X\right)$$

Where the model tries to predict the sentiment class using the feature vector X and $\sigma\left(z\right) = \frac{1}{1+e^{-z}}$, which is popularly known as the sigmoid function or logistic function. The main objective of this model is to search for an optimal value of θ such that the probability of the positive sentiment class is maximum when the feature vector *X* is for a positive movie review and small when it is for a negative movie review. The logistic function helps model the probability to describe the final prediction class. The optimal value of *θ* can be obtained by minimizing an appropriate cost/loss function using standard methods like gradient descent. Logistic regression is also popularly known as the *MaxEnt (maximum entropy) classifier*.

We now use our utility function `train_predict_model(...)` from our `model_evaluation_utils` module to build a logistic regression model on our training features and evaluate the model performance on our test features (Steps 4 and 5).

```
In [5]: # Logistic Regression model on BOW features
   ...: lr_bow_predictions = meu.train_predict_model(classifier=lr,
   ...: train_features=cv_train_features, train_labels=train_sentiments,
   ...: test_features=cv_test_features, test_labels=test_sentiments)
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...: predicted_labels=lr_bow_predictions,
   ...: classes=['positive', 'negative'])
```

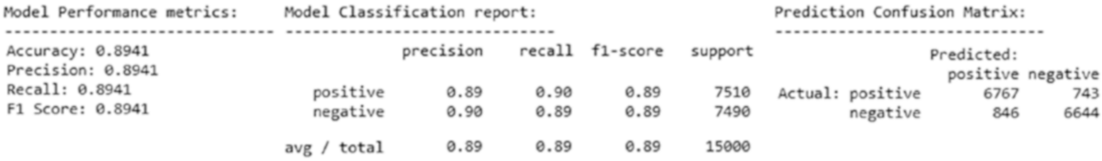


```
Model Performance metrics:     Model Classification report:                                 Prediction Confusion Matrix:
------------------------------ ------------------------------                              ------------------------------
Accuracy: 0.9049                              precision    recall  f1-score   support                    Predicted:
Precision: 0.9049                                                                                          positive negative
Recall: 0.9049                    positive       0.90      0.91      0.91      7510  Actual: positive        6809      701
F1 Score: 0.9049                  negative       0.91      0.90      0.90      7490          negative         725     6765

                               avg / total       0.90      0.90      0.90     15000
```

Figure 9-6. *Model performance metrics for logistic regression on Bag of Words features*

We get an overall F1-score and model accuracy of 90.5%, as depicted in Figure 9-6, which is excellent! We can now build a logistic regression model similarly on our TF-IDF features using the following snippet.

```
In [6]: # Logistic Regression model on TF-IDF features
   ...: lr_tfidf_predictions = meu.train_predict_model(classifier=lr,
   ...: train_features=tv_train_features, train_labels=train_sentiments,
   ...: test_features=tv_test_features, test_labels=test_sentiments)
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...: predicted_labels=lr_tfidf_predictions,
   ...: classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                      Prediction Confusion Matrix:
------------------------------ ------------------------------                    ------------------------------
Accuracy: 0.8941                            precision    recall  f1-score   support
Precision: 0.8941                                                                               Predicted:
Recall: 0.8941                                                                                    positive negative
F1 Score: 0.8941                  positive       0.89      0.90      0.89      7510  Actual: positive     6767      743
                                  negative       0.90      0.89      0.89      7490          negative      846     6644

                               avg / total       0.89      0.89      0.89     15000
```

Figure 9-7. *Model performance metrics for logistic regression on TF-IDF features*

We get an overall F1-score and model accuracy of 89%, as depicted in Figure 9-7, which is great but our previous model is slightly better. You can similarly use the support vector machine model estimator object `svm`, which we created earlier, and use the same snippet to train and predict using an SVM model. We obtained a maximum accuracy and F1-score of 90% with the SVM model (refer to the Jupyter notebook for step-by-step code snippets). Thus you can see how effective and accurate these supervised machine learning classification algorithms are in building a text sentiment classifier.

Newer Supervised Deep Learning Models

Deep learning has revolutionized the machine learning landscape over the last decade. In this section, we build some deep neural networks and train them on some advanced text features based on word embeddings to build a text sentiment classification system, similar to what we did in the previous section. Let's load the following necessary dependencies before we start our analysis.

```
In [7]: import gensim
   ...: import keras
   ...: from keras.models import Sequential
   ...: from keras.layers import Dropout, Activation, Dense
   ...: from keras.layers.normalization import BatchNormalization
   ...: from sklearn.preprocessing import LabelEncoder
Using TensorFlow backend.
```

So far, our models in Scikit-Learn directly accepted the sentiment class labels as `positive` and `negative` and internally performed these operations. However, for our deep learning models, we need to encode them explicitly. The following snippet helps us tokenize our movie reviews and convert the text-based sentiment class labels into one-hot encoded vectors (forms a part of Step 2).

```
In [8]: le = LabelEncoder()
   ...: num_classes=2
   ...: # tokenize train reviews & encode train labels
   ...: tokenized_train = [tn.tokenizer.tokenize(text)
   ...:                    for text in norm_train_reviews]
   ...: y_tr = le.fit_transform(train_sentiments)
   ...: y_train = keras.utils.to_categorical(y_tr, num_classes)
   ...: # tokenize test reviews & encode test labels
   ...: tokenized_test = [tn.tokenizer.tokenize(text)
   ...:                    for text in norm_test_reviews]
   ...: y_ts = le.fit_transform(test_sentiments)
   ...: y_test = keras.utils.to_categorical(y_ts, num_classes)
   ...:
   ...: # print class label encoding map and encoded labels
   ...: print('Sentiment class label map:', dict(zip(le.classes_,
        le.transform(le.classes_))))
   ...: print('Sample test label transformation:\n'+'-'*35,
   ...:       '\nActual Labels:', test_sentiments[:3], '\nEncoded Labels:',
              y_ts[:3],'\nOne hot encoded Labels:\n', y_test[:3])
Sentiment class label map: {'positive': 1, 'negative': 0}
Sample test label transformation:
-----------------------------------
Actual Labels: ['negative' 'positive' 'negative']
Encoded Labels: [0 1 0]
One hot encoded Labels:
 [[ 1.  0.]
  [ 0.  1.]
  [ 1.  0.]]
```

Thus, we can see from the preceding outputs how our sentiment class labels have been encoded into numeric representations, which in turn have been converted into one-hot encoded vectors. The feature engineering techniques we use in this section (Step 3) are slightly more advanced word vectorization techniques and are based on the concept of word embeddings. We use the Word2Vec and GloVe models to generate embeddings.

The Word2Vec model was built by Google and we covered this in detail in Chapter 4 under the section "Word Embeddings". We set the size parameter to 500 in this scenario, representing the feature vector size to be 512 for each word.

```
In [9]: # build word2vec model
   ...: w2v_num_features = 512
   ...: w2v_model = gensim.models.Word2Vec(tokenized_train, size=w2v_num_
        features, window=150, min_count=10, sample=1e-3)
```

We use the document word vector averaging scheme on this model from Chapter 4 to represent each movie review as an averaged vector of all the word vector representations for the different words in the review. The following function helps us compute averaged word vector representations for any corpus of text documents.

```
def averaged_word2vec_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)

    def average_word_vectors(words, model, vocabulary, num_features):
        feature_vector = np.zeros((num_features,), dtype="float64")
        nwords = 0.
        for word in words:
            if word in vocabulary:
                nwords = nwords + 1.
                feature_vector = np.add(feature_vector, model[word])
        if nwords:
            feature_vector = np.divide(feature_vector, nwords)
        return feature_vector

    features = [average_word_vectors(tokenized_sentence, model, vocabulary,
                 num_features) for tokenized_sentence in corpus]
    return np.array(features)
```

We can now use this function to generate averaged word vector representations on our two movie review datasets.

```
In [10]: # generate averaged word vector features from word2vec model
    ...: avg_wv_train_features = averaged_word2vec_vectorizer(corpus=
         tokenized_train, model=w2v_model, num_features= w2v_num_features)
    ...: avg_wv_test_features = averaged_word2vec_vectorizer(corpus=
         tokenized_test, model=w2v_model, num_features= w2v_num_features)
```

The GloVe model, which stands for Global Vectors, is an unsupervised model for obtaining word vector representations. Created at Stanford University, this model is trained on various corpora like Wikipedia, Common Crawl, and Twitter and corresponding pretrained word vectors are available and can be used for our analysis needs. Interested readers can refer to the original paper by Jeffrey Pennington, Richard Socher, and Christopher D. Manning, entitled "GloVe: Global Vectors for Word Representation" for more details. The spaCy library provided 300-dimensional word vectors trained on the Common Crawl corpus using the GloVe model. They provide a simple standard interface to get feature vectors of size 300 for each word as well as the averaged feature vector of a complete text document. The following snippet leverages spaCy to get the GloVe embeddings for our two datasets.

```
In [11]: # feature engineering with GloVe model
    ...: train_nlp = [tn.nlp_vec(item) for item in norm_train_reviews]
    ...: train_glove_features = np.array([item.vector for item in train_nlp])
    ...:
    ...: test_nlp = [tn.nlp_vec(item) for item in norm_test_reviews]
    ...: test_glove_features = np.array([item.vector for item in test_nlp])
```

You can check the feature vector dimensions for our datasets based on each of these models using the following code.

```
In [12]: print('Word2Vec model:> Train features shape:', avg_wv_train_
         features.shape, ' Test features shape:', avg_wv_test_features.shape)
    ...: print('GloVe model:> Train features shape:', train_glove_features.
         shape, ' Test features shape:', test_glove_features.shape)
Word2Vec model:> Train features shape: (35000, 512)  Test features shape:
(15000, 512)
GloVe model:> Train features shape: (35000, 300)  Test features shape:
(15000, 300)
```

We can see from the preceding output that, as expected, the Word2Vec model features are of size 500 and the GloVe features are of size 300.

We can now proceed to Step 4 of our classification system workflow, where we build and train a deep neural network on these features. We use a fully-connected four layer deep neural network (multi-layer perceptron or deep ANN) for our model. We do not count the input layer in any deep architecture, hence our model will consist of three hidden layers of 512 neurons or units and one output layer with two units, which will be used to predict a positive or negative sentiment based on the input layer features. Figure 9-8 depicts our deep neural network model for sentiment classification.

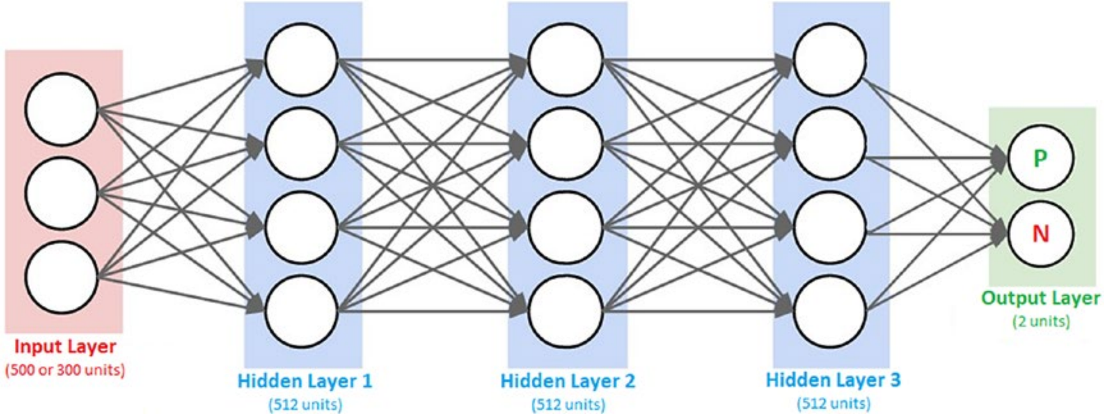


Figure 9-8. *Fully connected deep neural network model for sentiment classification*

We call this a fully connected deep neural network (DNN) because neurons or units in each pair of adjacent layers are fully pairwise connected. These networks are also known as deep artificial neural networks (ANNs) or multi-layer perceptrons (MLPs) since they have more than one hidden layer. The following function leverages Keras on top of TensorFlow to build the desired DNN model.

```
def construct_deepnn_architecture(num_input_features):
    dnn_model = Sequential()
    dnn_model.add(Dense(512, input_shape=(num_input_features,), kernel_
    initializer='glorot_uniform'))
    dnn_model.add(BatchNormalization())
    dnn_model.add(Activation('relu'))
    dnn_model.add(Dropout(0.2))

    dnn_model.add(Dense(512, kernel_initializer='glorot_uniform'))
    dnn_model.add(BatchNormalization())
    dnn_model.add(Activation('relu'))
    dnn_model.add(Dropout(0.2))
    dnn_model.add(Dense(512, kernel_initializer='glorot_uniform'))
    dnn_model.add(BatchNormalization())
    dnn_model.add(Activation('relu'))
    dnn_model.add(Dropout(0.2))

    dnn_model.add(Dense(2))
    dnn_model.add(Activation('softmax'))

    dnn_model.compile(loss='categorical_crossentropy', optimizer='adam',
                      metrics=['accuracy'])
    return dnn_model
```

From the preceding function, you can see that we accept a num_input_features parameter, which decides the number of units needed in the input layer (512 for Word2Vec and 300 for GloVe features). We build a Sequential model, which helps us in linearly stacking our hidden and output layers.

We use 512 units for all our hidden layers and the activation function relu indicates a rectified linear unit. This function is typically defined as $relu(x) = \max(0, x)$ where x is typically the input to a neuron. This is popularly known as the *ramp function* in electronics and electrical engineering. This function is preferred now as compared to the previously popular sigmoid function because it tries to solve the vanishing gradient problem. This problem occurs when $x > 0$ and as x increases, the gradient from sigmoids becomes very small (almost vanishing), but relu prevents this from happening. Besides this, it also helps in faster convergence of gradient descent.

Note that we also use a novel technique called *batch normalization*. Batch normalization is a technique for improving the performance and stability of neural networks. The key idea is to normalize the inputs of each layer in such a way that they have a mean output activation of 0 and standard deviation of 1. Remember that it is called *batch* normalization because during training, we normalize the activations of the previous layer for each batch, i.e., we apply a transformation such that we try to maintain the mean activation close to 0 and the standard deviation close to 1. This helps in regularization to some extent. Batch normalization tries to keep the distribution fed to a neural unit constant. This helps to keep gradients in proper bounds, which otherwise can lead to vanishing gradients, especially when using activation functions like sigmoid.

We also use regularization in the network in the form of Dropout layers. By adding a dropout rate of 0.2, it randomly sets 20% of the input feature units to 0 at each update during training the model. This form of regularization helps prevent overfitting the model.

The final output layer consists of two units with a softmax activation function. The softmax function is basically a generalization of the logistic function we saw earlier, which can be used to represent a probability distribution over *n* possible class outcomes. In our case, *n* = 2 where the class can either be *positive* or *negative* and the softmax probabilities will help us determine the same. The binary softmax classifier is also interchangeably known as the *binary logistic regression function*.

The compile(...) method is used to configure the learning or training process of the DNN model before we train it. This involves providing a cost or loss function in the loss parameter. This will be the goal or objective that the model will try to minimize. There are various loss functions based on the type of problem you want to solve, for example the mean squared error for regression and categorical cross-entropy for classification. Check out https://keras.io/losses/ for a list of possible loss functions. We will be using categorical_crossentropy which helps us minimize the error or loss from the softmax output. We need an optimizer for converging our model and minimizing the loss or error function. The gradient descent or stochastic gradient descent is a popular optimizer.

We use the adam optimizer which only requires first order gradients and very little memory. Adam also uses momentum where each update is based not only on the gradient computation of the current point, but also includes a fraction of the previous update. This helps in faster convergence. Interested readers can refer to the original paper from https://arxiv.org/pdf/1412.6980v8.pdf for further details on the adam optimizer. Finally, the metrics parameter specifies model performance metrics, which are used to evaluate the model when training (but not used to modify the training loss itself). Let's now build a DNN model based on our Word2Vec input feature representations for our training reviews.

```
In [13]: w2v_dnn = construct_deepnn_architecture(num_input_features=w2v_
num_features)
```

You can also visualize the DNN model architecture with the help of Keras, by using the following code. See Figure 9-9.

```
In [14]: from IPython.display import SVG
    ...: from keras.utils.vis_utils import model_to_dot
    ...:
    ...: SVG(model_to_dot(w2v_dnn, show_shapes=True, show_layer_
         names=False, rankdir='TB').create(prog='dot', format='svg'))
```

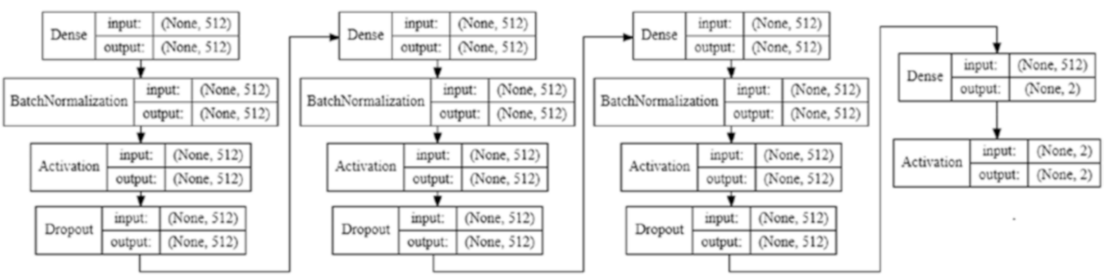


Figure 9-9. *Visualizing the DNN model architecture using Keras*

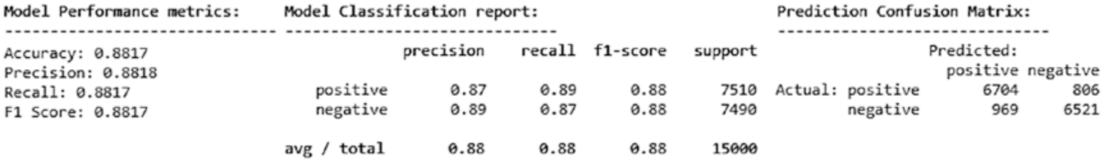
We now train our model on our training reviews dataset of Word2Vec features represented by avg_wv_train_features (Step 4). We use the fit(...) function from Keras for the training process. There are some parameters that you should be aware of. The epoch parameter indicates one complete forward and backward pass of all the training examples. The batch_size parameter indicates the total number of samples propagated through the DNN model at a time for one backward and forward pass for training the model and updating the gradient. Thus if you have 1,000 observations and your batch size is 100, each epoch will consist of 10 iterations, where 100 observations will be passed through the network at a time and the weights on the hidden layer units will be updated.

We also specify a validation_split of 0.1 to extract 10% of the training data and use it as a validation dataset for evaluating the performance at each epoch. The shuffle parameter shuffles the samples in each epoch when training the model.

```
In [18]: batch_size = 100
    ...: w2v_dnn.fit(avg_wv_train_features, y_train, epochs=10, batch_
         size=batch_size, shuffle=True, validation_split=0.1, verbose=1)
Train on 31500 samples, validate on 3500 samples
Epoch 1/10  31500/31500 - loss: 0.3378 - acc: 0.8598 - val_loss: 0.3114 - val_acc: 0.8714
Epoch 2/10 31500/31500 - loss: 0.2877 - acc: 0.8808 - val_loss: 0.2968 - val_acc: 0.8806
Epoch 3/10 31500/31500 - loss: 0.2766 - acc: 0.8854 - val_loss: 0.3043 - val_acc: 0.8726
Epoch 4/10 31500/31500 - loss: 0.2702 - acc: 0.8888 - val_loss: 0.2964 - val_acc: 0.8786
...
...
Epoch 9/10 31500/31500 - loss: 0.2456 - acc: 0.8964 - val_loss: 0.3180 - val_acc: 0.8680
Epoch 10/10 31500/31500 - loss: 0.2385 - acc: 0.9014 - val_loss: 0.3126 - val_acc: 0.8717
```

The preceding snippet tells us that we have trained our DNN model on the training data for 10 epochs with 100 as the batch size. We get a validation accuracy of close to 88%, which is quite good. It's time now to put our model to the real test! Let's evaluate our model performance on the test review Word2Vec features (Step 5).

```
In [19]: y_pred = w2v_dnn.predict_classes(avg_wv_test_features)
    ...: predictions = le.inverse_transform(y_pred)
    ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
    ...: predicted_labels=predictions, classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                              Prediction Confusion Matrix:
------------------------------ ------------------------------                              ------------------------------
Accuracy: 0.8817                            precision    recall  f1-score   support                   Predicted:
Precision: 0.8818                                                                                      positive negative
Recall: 0.8817                    positive       0.87      0.89      0.88      7510 Actual: positive       6704      806
F1 Score: 0.8817                  negative       0.89      0.87      0.88      7490         negative        969     6521

                               avg / total       0.88      0.88      0.88     15000
```

Figure 9-10. *Model performance metrics for deep neural networks on Word2Vec features*

The results in Figure 9-10 show us that we have obtained a model accuracy and F1-score of 88%, which is great! You can use a similar workflow and build and train a DNN model for our GloVe based features and evaluate the model performance. The following snippet depicts the workflow for Steps 4 and 5 of our text classification system blueprint.

```
# build DNN model
glove_dnn = construct_deepnn_architecture(num_input_features=300)
# train DNN model on GloVe training features
batch_size = 100
glove_dnn.fit(train_glove_features, y_train, epochs=5, batch_size=batch_size,
              shuffle=True, validation_split=0.1, verbose=1)
# get predictions on test reviews
y_pred = glove_dnn.predict_classes(test_glove_features)
predictions = le.inverse_transform(y_pred)
# Evaluate model performance
meu.display_model_performance_metrics(true_labels=test_sentiments,
predicted_labels=predictions, classes=['positive', 'negative'])
```

We obtained an overall model accuracy and F1-score of 86% with the GloVe features, which is still good but not better than what we obtained using our Word2Vec features. You can refer to the `Sentiment Analysis - Supervised.ipynb` Jupyter notebook to see the step-by-step output obtained for this code. This concludes our discussion on building text sentiment classification systems leveraging newer deep learning models and methodologies. Onwards to learning about advanced deep learning models!

Advanced Supervised Deep Learning Models

We used fully connected deep neural network and word embeddings in the previous section. Another new and interesting approach to supervised deep learning is the use of recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) which also considers the sequence of data (words, events and so on). These are more advanced models than your regular fully connected deep networks and usually take more time to train. We leverage Keras on top of TensorFlow and try to build a LSTM-based classification model and use word embeddings as our features. You can refer to the Jupyter notebook titled `Sentiment Analysis - Advanced Deep Learning.ipynb` for an interactive experience.

We work on our normalized and preprocessed train and test review datasets, `norm_train_reviews` and `norm_test_reviews`, which we created in our previous analyses. Assuming you have them loaded, we will first tokenize these datasets such that each text review is decomposed into its corresponding tokens (workflow Step 2).

```
In [1]: tokenized_train = [tn.tokenizer.tokenize(text) for text in norm_
                           train_reviews]
   ...: tokenized_test = [tn.tokenizer.tokenize(text) for text in norm_
                          test_reviews]
```

For feature engineering (Step 3), we create word embeddings. However, we will create them using Keras instead of using prebuilt ones like Word2Vec or GloVe. Word embeddings tend to vectorize text documents into fixed sized vectors such that these vectors try to capture contextual and semantic information.

To generate embeddings, we use the `Embedding` layer from Keras, which requires documents to be represented as tokenized and numeric vectors. We already have tokenized text vectors in our `tokenized_train` and `tokenized_text` variables. However, we would need to convert them into numeric representations. Besides this, we would also need the vectors to be of uniform size even though the tokenized text reviews will be of variable length due to the difference in number of tokens in each review. For this, one strategy is to take the length of the longest review (with maximum number of tokens/words) and set it as the vector size. Let's call this `max_len`. Reviews of shorter length can be padded with a `PAD` term in the beginning to increase their length to `max_len`.

We would need to create a word to index vocabulary mapping for representing each tokenized text review in a numeric form. Note you also need to create a numeric mapping for the padding term, which we will call `PAD_INDEX`, and assign it the numeric index of `0`. For unknown terms, in case they are encountered later in the test dataset or newer, previously unseen reviews, we would need to assign them to some index too. This is because we will vectorize, engineer, and build models only on the training data. Hence, if a new term should come up (which was originally not a part of the model training), we will consider it as an out of vocabulary (OOV) term and assign it to a constant index (we name this term `NOT_FOUND_INDEX` and assign it the index of `vocab_size+1`).

The following snippet helps us create this vocabulary from our `tokenized_train` corpus of training text reviews.

```
In [2]: from collections import Counter
   ...:
   ...: # build word to index vocabulary
   ...: token_counter = Counter([token for review in tokenized_train for
        token in review])
   ...: vocab_map = {item[0]: index+1
                    for index, item in enumerate(dict(token_counter).items())}
   ...: max_index = np.max(list(vocab_map.values()))
   ...: vocab_map['PAD_INDEX'] = 0
   ...: vocab_map['NOT_FOUND_INDEX'] = max_index+1
   ...: vocab_size = len(vocab_map)
   ...: # view vocabulary size and part of the vocabulary map
   ...: print('Vocabulary Size:', vocab_size)
   ...: print('Sample slice of vocabulary map:', dict(list(vocab_map.
        items())[10:20]))
Vocabulary Size: 82358
Sample slice of vocabulary map: {'martyrdom': 6, 'palmira': 7, 'servility': 8,
'gardening': 9, 'melodramatically': 73505, 'renfro': 41282, 'carlin': 41283,
'overtly': 41284, 'rend': 47891, 'anticlimactic': 51}
```

In this case, we used all the terms in our vocabulary. You can easily filter and use more relevant terms here (based on their frequency) by using the `most_common(count)` function from `Counter` and taking the first `count` terms from the list of unique terms in the training corpus. We now encode the tokenized text reviews based on the `vocab_map`. Besides this, we also encode the text sentiment class labels into numeric representations.

```
In [3]: from keras.preprocessing import sequence
   ...: from sklearn.preprocessing import LabelEncoder
   ...:
   ...: # get max length of train corpus and initialize label encoder
   ...: le = LabelEncoder()
   ...: num_classes=2 # positive -> 1, negative -> 0
   ...: max_len = np.max([len(review) for review in tokenized_train])
   ...:
```

```
   ...: ## Train reviews data corpus
   ...: # Convert tokenized text reviews to numeric vectors
   ...: train_X = [[vocab_map[token] for token in tokenized_review]
                       for tokenized_review in tokenized_train]
   ...: train_X = sequence.pad_sequences(train_X, maxlen=max_len) # pad
   ...: ## Train prediction class labels
   ...: # Convert text sentiment labels (negative\positive) to binary
        encodings (0/1)
   ...: train_y = le.fit_transform(train_sentiments)
   ...:
   ...: ## Test reviews data corpus
   ...: # Convert tokenized text reviews to numeric vectors
   ...: test_X = [[vocab_map[token] if vocab_map.get(token) else vocab_
        map['NOT_FOUND_INDEX']
   ...:            for token in tokenized_review]
   ...:               for tokenized_review in tokenized_test]
   ...: test_X = sequence.pad_sequences(test_X, maxlen=max_len)
   ...: ## Test prediction class labels
   ...: # Convert text sentiment labels (negative\positive) to binary
        encodings (0/1)
   ...: test_y = le.transform(test_sentiments)
   ...:
   ...: # view vector shapes
   ...: print('Max length of train review vectors:', max_len)
   ...: print('Train review vectors shape:', train_X.shape,
               ' Test review vectors shape:', test_X.shape)

Max length of train review vectors: 1442
Train review vectors shape: (35000, 1442)  Test review vectors shape:
(15000, 1442)
```

From the preceding code snippet and the output, it is clear that we encoded each text review into a numeric sequence vector such that the size of each review vector is 1,442, which is basically the maximum length of reviews from the training dataset. We pad shorter reviews and truncate extra tokens from longer reviews such that the shape of each review is constant, as depicted in the output. We can now proceed to Step 3 and a part of Step 4 of the classification workflow by introducing the Embedding layer and coupling it with the deep network architecture based on LSTMs.

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, Dropout, SpatialDropout1D
from keras.layers import LSTM

EMBEDDING_DIM = 128 # dimension for dense embeddings for each token
LSTM_DIM = 64 # total LSTM units

model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=EMBEDDING_DIM, input_
length=max_len))
model.add(SpatialDropout1D(0.2))
model.add(LSTM(LSTM_DIM, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation="sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

The Embedding layer helps us generate the word embeddings from scratch. This layer is also initialized with some weights and is updated based on our optimizer, similar to weights on the neuron units in other layers when the network tries to minimize the loss in each epoch. Thus, the embedding layer tries to optimize its weights such that we get the best word embeddings that will generate minimum error in the model and capture semantic similarity and relationships among words. How do we get the embeddings? Let's say we have a review with three terms ['movie', 'was', 'good'] and a vocab_map consisting of word to index mappings for 82,358 words. The word embeddings are generated somewhat similar to what's shown in Figure 9-11.

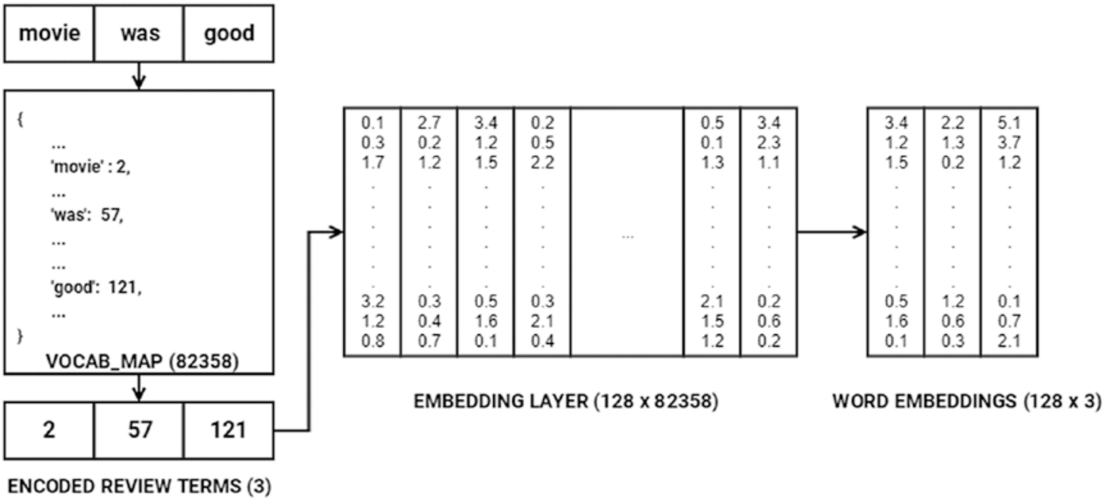


***Figure 9-11.** Understanding how word embeddings are generated*

Based on our model architecture, the `Embedding` layer takes in three parameters:

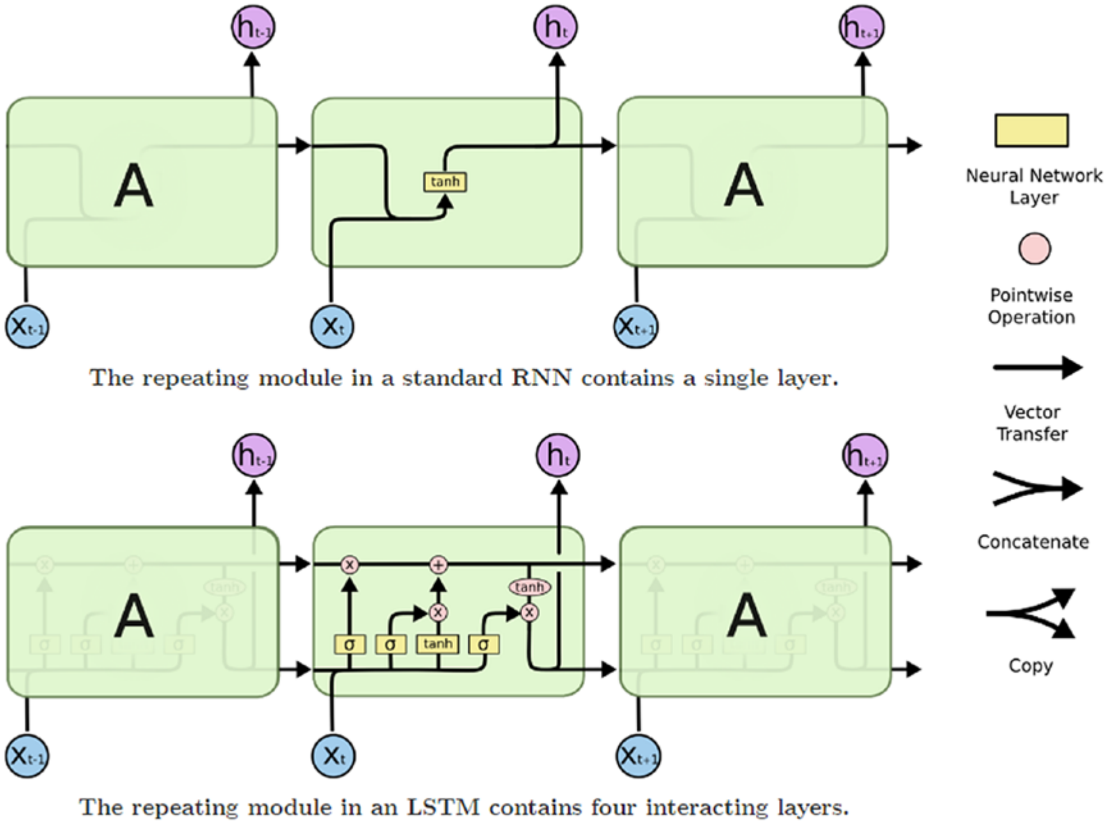
- `input_dim`, which is equal to the vocabulary size (`vocab_size`) of 82,358
- `output_dim`, which is 128, representing the dimension of dense embedding (depicted by rows in the `embedding` layer in Figure 9-11)
- `input_len`, which specifies the length of the input sequences (movie review sequence vectors), which is 1,442

In the example depicted in Figure 9-11, since we have one review, the dimension is (1, 3). This review is converted into a numeric sequence (2, 57, 121) based on the `VOCAB_MAP`. Then the specific columns representing the indices in the review sequence are selected from the `embedding` layer (vectors at column indices 2, 57, and 121) to generate the final word embeddings. This gives us an embedding vector of dimension (1, 128, 3) also represented as (1, 3, 128) when each row is represented based on each sequence word embedding vector. Many deep learning frameworks like Keras represent the embedding dimensions as (m, n) where m represents all the unique terms in our vocabulary (82,358) and n represents the `output_dim`, which is 128 in this case. Consider a transposed version of the layer depicted in Figure 9-11 and you are good to go!

If you have the encoded review terms sequence vector represented in one-hot encoded format (3, 82358) and do a matrix multiplication with the `embedding` layer represented as (82358, 128), where each row represents the embedding for a word in the vocabulary, you will directly obtain the word embeddings for the review sequence vector as (3, 128). The weights in the embedding layer are updated and optimized in each epoch based on the input data when propagated through the whole network, like we mentioned earlier such that overall loss and error is minimized to get maximum model performance.

These dense word embeddings are then passed to the LSTM layer having 64 units. We introduced the LSTM architecture briefly in Chapter 1. LSTMs try to overcome the shortcomings of RNN models, especially with regard to handling long-term dependencies and problems that occur when the weight matrix associated with the units (neurons) become too small (leading to vanishing gradient) or too large (leading to exploding gradient). These architectures are more complex than regular deep networks and going into detailed internals and math concepts are out of the current scope, but we will try to cover the essentials here without making it math heavy.

Readers interested in researching the internals of LSTMs can check out the original paper which inspired it all, by Hochreiter, S., and Schmidhuber, J. entitled, "Long Short-Term Memory," from Neural *Computation*, 9(8), 1735-1780. We depict the basic architecture of RNNs and compare it to LSTMs in Figure 9-12.



The repeating module in a standard RNN contains a single layer.

The repeating module in an LSTM contains four interacting layers.

Figure 9-12. *Basic structure of RNN and LSTM units (Source: Christopher Olah's blog: colah.github.io)*

The RNN units usually have a chain of repeating modules (this happens when we unroll the loop) so that the module has a simple structure of maybe one layer with the `tanh` activation. LSTMs are also a special type of RNN, with a similar structure, but the LSTM unit has four neural network layers instead of just one. The detailed architecture of the LSTM cell is shown in Figure 9-13.

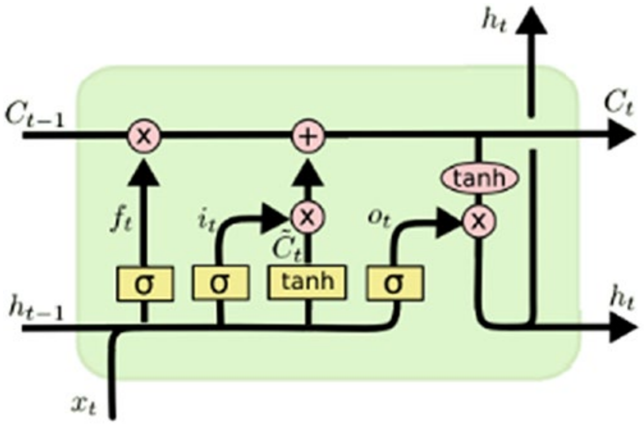


***Figure 9-13.** Detailed architecture of an LSTM cell (Source: Christopher Olah's blog: colah.github.io)*

The notation *t* indicates one time step, *C* depicts the cell states, and *h* indicates the hidden states. The gates $i, f, o, and\ \overset{\vee}{C_t}$ help remove or add information to the cell state. The gates *i, f, and o* represent the *input, output,* and *forget* gates, respectively. Each of them is modulated by the sigmoid layer, which outputs numbers from 0 to 1 controlling how much of the output from these gates should pass. This protects and controls the cell state. The detailed workflow of how information flows through the LSTM cell is depicted in Figure 9-14 in four steps.

1. The first step talks about the forget gate layer *f* which helps us decide what information we should throw away from the cell state. This is done by looking at the previous hidden state h_{t-1} and current inputs x_t as depicted in the equation. The sigmoid layer helps control how much of this should be kept or forgotten.

2. The second step depicts the input gate layer *i* which helps decide what information will be stored in the current cell state. The sigmoid layer in the input gate helps decide which values will be updated based on h_{t-1} & x_t. The tanh layer helps create a vector of the new candidate values $\overset{\vee}{C_t}$, based on h_{t-1} & x_t, which can be added to the current cell state. Thus the tanh layer creates the values and the input gate with sigmoid layer helps choose which values should be updated.

3. The third step involves updating the old cell state C_{t-1} to the new cell state C_t by leveraging what we obtained in the first two steps. We multiply the old cell state by the forget gate ($f_t \times C_{t-1}$) and then add the new candidate values scaled by the input gate to sigmoid layer $\left(i_t \times \check{C}_t\right)$.

4. The fourth and final step helps us decide what the final output should be, which is basically a filtered version of our cell state. The output gate with the sigmoid layer o helps us select which parts of the cell state will pass to the final output. This is multiplied with the cell state values when passed through the tanh layer to give us the final hidden state values $h_t = o_t \times \tanh\left(\check{C}_t\right)$.

These steps are depicted in Figure 9-14 with necessary annotations and equations. I want to thank our good friend Christopher Olah for providing us with detailed information as well as the images for depicting the internal workings of LSTM networks. We recommend checking out Christopher's blog at http://colah.github.io/posts/2015-08-Understanding-LSTMs for more details. A shout out also goes to Edwin Chen, for explaining RNNs and LSTMs in an easy-to-understand format. We also recommend referring to Edwin's blog at http://blog.echen.me/2017/05/30/exploring-lstms for information on the workings of RNNs and LSTMs.

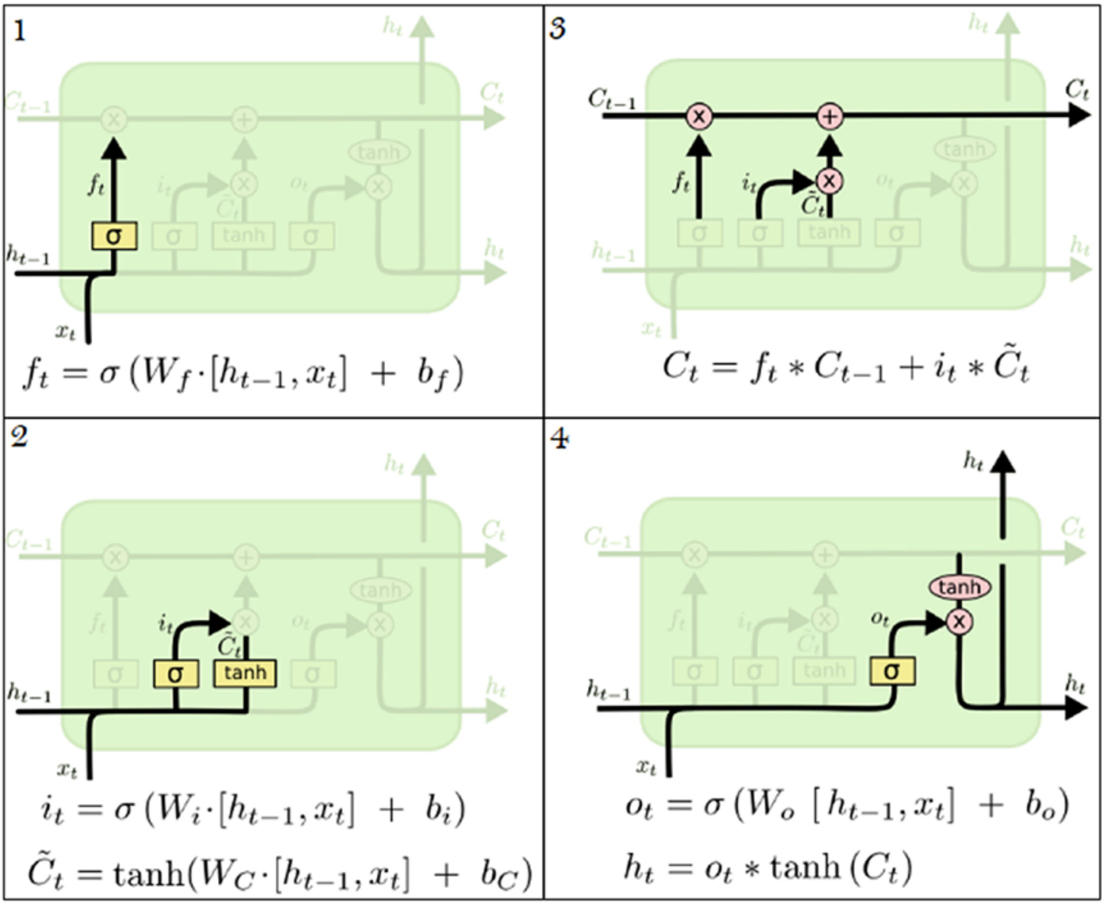

1
$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$
2
$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$
$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
3
$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
4
$o_t = \sigma\left(W_o\ [h_{t-1}, x_t] + b_o\right)$
$h_t = o_t * \tanh\left(C_t\right)$
C_{t-1}
C_t
h_{t-1}
h_t
x_t
f_t
i_t
o_t
$\tilde{C}_t$
σ
tanh

Figure 9-14. *Walkthrough of data flow in an LSTM cell (Source: Christopher Olah's blog: colah.github.io)*

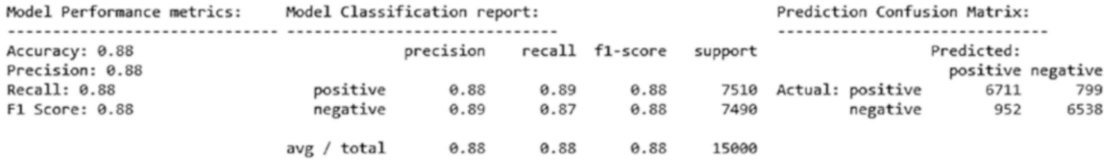
The final layer in our deep network is the Dense layer with 1 unit and the sigmoid activation function. We basically use the binary_crossentropy function with the adam optimizer, since this is a binary classification problem and the model will ultimately predict a 0 or a 1, which we can decode back to a negative or positive sentiment prediction with our label encoder.

You can also use the categorical_crossentropy loss function here, but you would need to then use a Dense layer with two units instead with a softmax function. Now that our model is compiled and ready, we can head on to Step 4 of our classification workflow—training the model. We use a similar strategy from our previous deep network models, where we train our model on the training data with five epochs, batch size of 100 reviews, and a 10% validation split of training data to measure validation accuracy.

```
In [4]: batch_size = 100
   ...: model.fit(train_X, train_y, epochs=5, batch_size=batch_size,
   ...: shuffle=True, validation_split=0.1, verbose=1)
Train on 31500 samples, validate on 3500 samples
Epoch 1/5 31500/31500 - 2491s - loss: 0.4081 - acc: 0.8184 - val_loss:
0.3006 - val_acc: 0.8751
Epoch 2/5 31500/31500 - 2489s - loss: 0.2253 - acc: 0.9158 - val_loss:
0.3209 - val_acc: 0.8780
Epoch 3/5 31500/31500 - 2656s - loss: 0.1431 - acc: 0.9493 - val_loss:
0.3483 - val_acc: 0.8671
Epoch 4/5 31500/31500 - 2604s - loss: 0.1023 - acc: 0.9658 - val_loss:
0.3803 - val_acc: 0.8729
Epoch 5/5 31500/31500 - 2701s - loss: 0.0694 - acc: 0.9761 - val_loss:
0.4430 - val_acc: 0.8706
```

Training LSTMs on CPU is notoriously slow and, as you can see, my model took approximately 3.6 hours to train just five epochs on an i5 3rd Gen Intel CPU with 8GB of memory. Of course, a cloud-based environment like Google Cloud Platform or AWS on GPU took me approximately less than an hour to train the same model. So I recommend you to choose a GPU-based deep learning environment, especially when working with RNNs or LSTM-based network architectures. Based on the preceding output, we can see that with just five epochs, we have decent validation accuracy. Time to put our model to the test! Let's see how well it predicts the sentiment for our test reviews and use the same model evaluation framework we used in our previous models (Step 5).

```
In [5]: # predict sentiments on test data
   ...: pred_test = model.predict_classes(test_X)
   ...: predictions = le.inverse_transform(pred_test.flatten())
   ...: # evaluate model performance
   ...: meu.display_model_performance_metrics(true_labels=test_sentiments,
   ...:                    predicted_labels=predictions,
                           classes=['positive', 'negative'])
```



```
Model Performance metrics:     Model Classification report:                          Prediction Confusion Matrix:
------------------------------ ------------------------------                        ------------------------------
Accuracy: 0.88                              precision    recall  f1-score   support                   Predicted:
Precision: 0.88                                                                                         positive negative
Recall: 0.88                      positive       0.88      0.89      0.88      7510  Actual: positive       6711      799
F1 Score: 0.88                    negative       0.89      0.87      0.88      7490          negative        952     6538

                               avg / total       0.88      0.88      0.88     15000
```

Figure 9-15. *Model performance metrics for LSTM-based deep learning model on word embeddings*

The results depicted in Figure 9-15 show us that we obtained a model accuracy and F1-score of 88%, which is quite good! With more quality data, you can expect to get even better results. Try experimenting with different architectures and see if you get better results!

Analyzing Sentiment Causation

We built both supervised and unsupervised models to predict the sentiment of movie reviews based on the review text content. While feature engineering and modeling is definitely the need of the hour, you also need to know how to analyze and interpret the root cause behind how model predictions work. In this section, we analyze sentiment causation. The idea is to find the root cause or key factors causing positive or negative sentiment. The first area of focus is model interpretation, where we try to understand, interpret, and explain the mechanics behind predictions made by our classification models. The second area of focus is to apply topic modeling and extract key topics from positive and negative sentiment reviews.

Interpreting Predictive Models

One of the challenges with machine learning models is the transition from a pilot or proof-of-concept phase to the production phase. Business and key stakeholders often perceive machine learning models as complex black boxes and pose the question, why should I trust your model? Explaining the complex mathematical or theoretical concepts doesn't serve the purpose. Is there some way that we can explain these models in an easy-to-interpret manner? This topic in fact has gained extensive attention very recently in 2016. Refer to the original research paper by M.T. Ribeiro, S. Singh, and

C. Guestrin titled "Why Should I Trust You?: Explaining the Predictions of Any Classifier," from https://arxiv.org/pdf/1602.04938.pdf to understand more about model interpretation and the LIME framework.

There are various ways to interpret the predictions made by our predictive sentiment classification models. We want to understand why a positive review was correctly predicted as having a positive sentiment or a negative review as having a negative sentiment. Besides this, no model is a 100% accurate, so we also want to understand the reason for misclassifications or wrong predictions. The code used in this section is available in the the Jupyter notebook named Sentiment Causal Analysis - Model Interpretation.ipynb for an interactive experience.

Let's first build a basic text classification pipeline for the model that worked best for us so far. This is the logistic regression model based on the Bag of Words feature model. We leverage the pipeline module from Scikit-Learn to build this machine learning pipeline using the following code.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

# build BOW features on train reviews
cv = CountVectorizer(binary=False, min_df=0.0, max_df=1.0, ngram_
range=(1,2))
cv_train_features = cv.fit_transform(norm_train_reviews)
# build Logistic Regression model
lr = LogisticRegression()
lr.fit(cv_train_features, train_sentiments)

# Build Text Classification Pipeline
lr_pipeline = make_pipeline(cv, lr)

# save the list of prediction classes (positive, negative)
classes = list(lr_pipeline.classes_)
```

We build our model based on norm_train_reviews, which contains the normalized training reviews that we used in all our earlier analyses. Now that we have our classification pipeline ready, you can deploy the model by using pickle or joblib to save the classifier and feature objects, similar to what we discussed in the "Model

Deployment" section in Chapter 5. Assuming our pipeline is in production, how do we use it for new movie reviews? Let's try to predict the sentiment for two new sample reviews (which were not used in training the model).

```
# normalize sample movie reviews
new_corpus = ['The Lord of the Rings is an Excellent movie!',
              'I didn\'t like the recent movie on TV. It was NOT good and a
              waste of time!']
norm_new_corpus = tn.normalize_corpus(new_corpus, stopwords=stop_words)
norm_new_corpus
```

```
['lord rings excellent movie', 'not like recent movie tv not good waste time']
```

```
# predict movie review sentiment
lr_pipeline.predict(norm_new_corpus)
```

```
array(['positive', 'negative'], dtype=object)
```

Our classification pipeline predicts the sentiment of both reviews correctly! Also closely observe the second sentence—it can handle negation, which is a desired quality. This is a good start, but how do we interpret the model predictions? One way is to use the model prediction class probabilities as a measure of confidence. You can use the following code to get the prediction probabilities for our sample reviews.

```
In [4]: pd.DataFrame(lr_pipeline.predict_proba(norm_new_corpus),
columns=classes)
Out[4]:
   negative  positive
0  0.217474  0.782526
1  0.912649  0.087351
```

Thus, we can say that the first movie review has a prediction confidence or probability of 78% to have positive sentiment as compared to the second movie review with a 91% probability to have negative sentiment.

Let's now kick it up a notch. Instead of playing around with toy examples, we run the same analysis on actual reviews from the `test_reviews` dataset (we use `norm_test_reviews`, which has the normalized text reviews). Besides prediction probabilities, we use the `skater` framework for easy interpretation of the model decisions. You need to load the following dependencies from the skater package first. We also define a helper function, which takes in a document index, a corpus, its response predictions, and an explainer object and helps with the model interpretation analysis.

```
from skater.core.local_interpretation.lime.lime_text import
LimeTextExplainer

explainer = LimeTextExplainer(class_names=classes)
# helper function for model interpretation
def interpret_classification_model_prediction(doc_index, norm_corpus,
corpus, prediction_labels, explainer_obj):
    # display model prediction and actual sentiments
    print("Test document index: {index}\nActual sentiment: {actual}
                                   \nPredicted sentiment: {predicted}"
      .format(index=doc_index, actual=prediction_labels[doc_index],
              predicted=lr_pipeline.predict([norm_corpus[doc_index]])))
    # display actual review content
    print("\nReview:", corpus[doc_index])
    # display prediction probabilities
    print("\nModel Prediction Probabilities:")
    for probs in zip(classes, lr_pipeline.predict_proba([norm_corpus[doc_
    index]])[0]):
        print(probs)
    # display model prediction interpretation
    exp = explainer.explain_instance(norm_corpus[doc_index],
                                     lr_pipeline.predict_proba, num_
                                     features=10, labels=[1])
    exp.show_in_notebook()
```

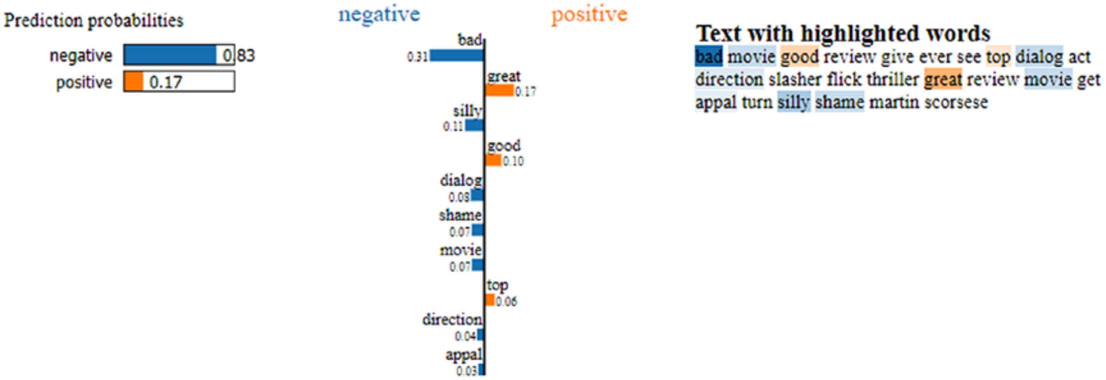
The preceding snippet leverages skater to explain our text classifier to analyze its decision making process in an easy-to-interpret form. Even though the model might be a complex one from a global perspective, it is easier to explain and approximate the model behavior on local instances. This is done by learning the model around the vicinity of the data point of interest *X* by sampling instances around *X* and assigning weights based on their proximity to *X*. Thus, these locally learned linear models help explain complex models in an easier way with class probabilities and contribution of top features to the class probabilities, which aid in the decision making process.

Let's take a movie review from our test dataset where both the actual and predicted sentiment is negative and analyze it with the helper function we created in the preceding snippet.

```
In [6]: doc_index = 100
   ...: interpret_classification_model_prediction(doc_index=doc_index,
        corpus=norm_test_reviews, corpus=test_reviews, prediction_
        labels=test_sentiments, explainer_obj=explainer)
```

```
Test document index: 100
Actual sentiment: negative
Predicted sentiment: ['negative']
```

```
Review: Worst movie, (with the best reviews given it) I've ever seen. Over
the top dialog, acting, and direction. more slasher flick than thriller.
With all the great reviews this movie got I'm appalled that it turned out
so silly. shame on you martin scorsese

Model Prediction Probabilities:
('negative', 0.827942236512913)
('positive', 0.17205776348708696)
```



bad movie good review give ever see top dialog act direction slasher flick thriller great review movie get appal turn silly shame martin scorsese

Figure 9-16. *Model interpretation for our classification model's correct prediction for a negative review*

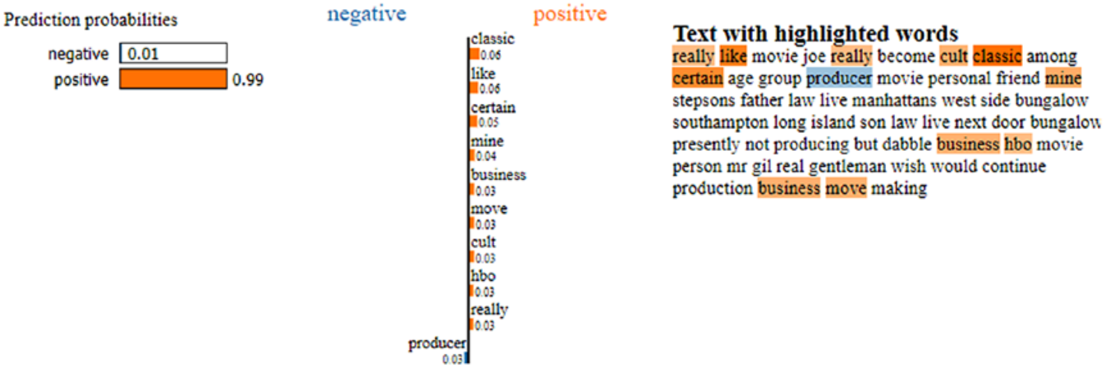
The results depicted in Figure 9-16 show us the class prediction probabilities and the top 10 features that contributed the maximum to the prediction decision-making process. These key features are also highlighted in the normalized movie review text. Our model performs quite well in this scenario and we can see the key features that contributed to the negative sentiment of this review including bad, silly, dialog, and shame, which make sense. Besides this, the word "great" contributed the maximum to the positive probability of 0.17 and in fact if we had removed this word from our review text, the positive probability would have dropped significantly.

The following code runs a similar analysis on a test movie review with both actual and predicted sentiment of the positive value.

```
In [7]: doc_index = 2000
   ...: interpret_classification_model_prediction(doc_index=doc_index,
        corpus=norm_test_reviews, corpus=test_reviews,prediction_labels=
                                  test_sentiments, explainer_obj=explainer)
```

```
Test document index: 2000
Actual sentiment: positive
Predicted sentiment: ['positive']
```

```
Review: I really liked the Movie "JOE." It has really become a cult
classic among certain age groups.<br /><br />The Producer of this movie is
a personal friend of mine. He is my Stepsons Father-In-Law. He lives in
Manhattan's West side, and has a Bungalow. in Southampton, Long Island. His
son-in-law live next door to his Bungalow.<br /><br />Presently, he does
not do any Producing, But dabbles in a business with HBO movies.<br />
<br />As a person, Mr. Gil is a real gentleman and I wish he would have
continued in the production business of move making.

Model Prediction Probabilities:
('negative', 0.014587305153566432)
('positive', 0.9854126948464336)
```



Text with highlighted words

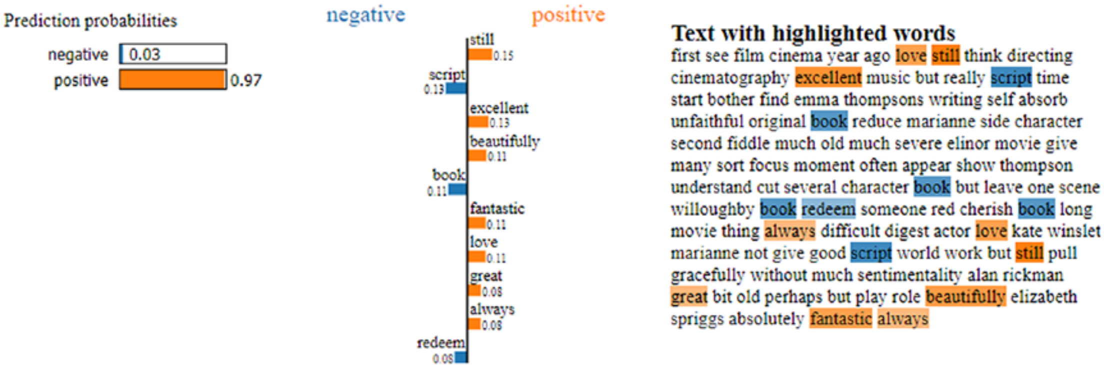
really like movie joe really become cult classic among certain age group producer movie personal friend mine stepsons father law live manhattans west side bungalow southampton long island son law live next door bungalow presently not producing but dabble business hbo movie person mr gil real gentleman wish would continue production business move making

***Figure 9-17.** Model interpretation for our classification model's correct prediction for a positive review*

The results depicted in Figure 9-17 show us the top features responsible for the model making a decision of predicting this review as positive. Based on the content, the reviewer really liked this movie and it was a real cult classic among certain age groups. In our final analysis, we look at the model interpretation of an example where the model makes a wrong prediction.

```
In [8]: doc_index = 347
   ...: interpret_classification_model_prediction(doc_index=doc_index,
        corpus=norm_test_reviews, corpus=test_reviews, prediction_labels=
        test_sentiments, explainer_obj=explainer)

Test document index: 347
Actual sentiment: negative
Predicted sentiment: ['positive']

Review: When I first saw this film in cinema 11 years ago, I loved it.
I still think the directing and cinematography are excellent, as is
the music. But it's really the script that has over the time started to
bother me more and more. I find Emma Thompson's writing self-absorbed
and unfaithful to the original book; she has reduced Marianne to a side-
character, a second fiddle to her much too old, much too severe Elinor -
she in the movie is given many sort of 'focus moments', and often they
appear to be there just to show off Thompson herself.<br /><br />I do
```

```
understand her cutting off several characters from the book, but leaving
out the one scene where Willoughby in the book is redeemed? For someone
who red and cherished the book long before the movie, those are the things
always difficult to digest.<br /><br />As for the actors, I love Kate
Winslet as Marianne. She is not given the best script in the world to work
with but she still pulls it up gracefully, without too much sentimentality.
Alan Rickman is great, a bit old perhaps, but he plays the role
beautifully. And Elizabeth Spriggs, she is absolutely fantastic as always.

Model Prediction Probabilities:
('negative', 0.028707732768304406)
('positive', 0.9712922672316956)
```



Text with highlighted words

first see film cinema year ago love still think directing cinematography excellent music but really script time start bother find emma thompsons writing self absorb unfaithful original book reduce marianne side character second fiddle much old much severe elinor movie give many sort focus moment often appear show thompson understand cut several character book but leave one scene willoughby book redeem someone red cherish book long movie thing always difficult digest actor love kate winslet marianne not give good script world work but still pull gracefully without much sentimentality alan rickman great bit old perhaps but play role beautifully elizabeth spriggs absolutely fantastic always

Figure 9-18. *Model interpretation for our classification model's incorrect prediction*

The preceding output tells us that our model predicted the movie review indicating a positive sentiment when in fact the actual sentiment label is negative for the same review. The results in Figure 9-18 tell us that the reviewer shows signs of positive sentiment in the movie review, especially in parts where he/she tells us that “I loved it. I still think the directing and cinematography are excellent, as is the music… Alan Rickman is great, a bit old perhaps, but he plays the role beautifully. And Elizabeth Spriggs, she is absolutely fantastic as always.” The feature words from the same have been depicted in the top features contributing to positive sentiment.

The model interpretation also correctly identifies the aspects of the review contributing to negative sentiment, such as "But it's really the script that has over time started to bother me more and more." Hence, this is one of the more complex reviews because it indicates positive and negative sentiment. The final interpretation is in the reader's hands. You can now use this same framework to interpret your own classification models and understand where your model might be performing well and where it might need improvements!

Analyzing Topic Models

Another way of analyzing key terms, concepts, or topics responsible for sentiment is to use a different approach known as *topic modeling*. We covered some basics of topic modeling in the section titled "Topic Models" under "Feature Engineering on Text Data" in Chapter 4. The main aim of topic models is to extract and depict key topics or concepts that are otherwise latent and not very prominent in huge corpora of text documents. We saw the use of Latent Dirichlet Allocation (LDA) and Non-Negative Matrix Factorization (NMF) for topic modeling in Chapter 6. In this section, we use Non-Negative Matrix Factorization. Refer to the Jupyter notebook titled `Sentiment Causal Analysis - Topic Models.ipynb` for an interactive experience.

The first step in this analysis is to combine all our normalized train and test reviews and separate these reviews into positive and negative sentiment reviews. Once we do this, we will extract features from these two datasets using the TF-IDF feature vectorizer. The following snippet helps us achieve this.

```
In [11]: from sklearn.feature_extraction.text import TfidfVectorizer
    ...:
    ...: # consolidate all normalized reviews
    ...: norm_reviews = norm_train_reviews+norm_test_reviews
    ...: # get tf-idf features for only positive reviews
    ...: positive_reviews = [review for review, sentiment in zip(norm_
         reviews, sentiments) if sentiment == 'positive']
    ...: ptvf = TfidfVectorizer(use_idf=True, min_df=0.02, max_df=0.75,
                                ngram_range=(1,2), sublinear_tf=True)
    ...: ptvf_features = ptvf.fit_transform(positive_reviews)
    ...: # get tf-idf features for only negative reviews
    ...: negative_reviews = [review for review, sentiment in zip(norm_
         reviews, sentiments) if sentiment == 'negative']
    ...: ntvf = TfidfVectorizer(use_idf=True, min_df=0.02, max_df=0.75,
                                ngram_range=(1,2), sublinear_tf=True)
    ...: ntvf_features = ntvf.fit_transform(negative_reviews)
    ...: # view feature set dimensions
    ...: print(ptvf_features.shape, ntvf_features.shape)

(25000, 933) (25000, 925)
```

From the preceding output dimensions, you can see that we have filtered out a lot of the features we used previously when building our classification models by making `min_df` to be 0.02 and `max_df` to be 0.75. This is to speed up the topic modeling process and remove features that either occur too much or not very often. Let's now import the necessary dependencies for the topic modeling process.

```
In [12]: import pyLDAvis
    ...: import pyLDAvis.sklearn
    ...: from sklearn.decomposition import NMF
    ...: import topic_model_utils as tmu
    ...:
    ...: pyLDAvis.enable_notebook()
    ...: total_topics = 10
```

The `NMF` class from Scikit-Learn helps us do the topic modeling. We also use pyLDAvis to build interactive visualizations of topic models. The core principle behind Non-Negative Matrix Factorization (NNMF) is to apply matrix decomposition (similar to SVD) to a non-negative feature matrix *X* so that the decomposition can be represented as $X \approx WH$, where *W and H* are both non-negative matrices which, if multiplied, should approximately reconstruct the feature matrix *X*. A cost function like L2 norm can be used to get this approximation. Let's apply NNMF to get 10 topics from our positive sentiment reviews.

```
# build topic model on positive sentiment review features
pos_nmf = NMF(n_components=total_topics, solver='cd', max_iter=500,
              random_state=42, alpha=.1, l1_ratio=.85)
pos_nmf.fit(ptvf_features)
```

```
# extract features and component weights
pos_feature_names = np.array(ptvf.get_feature_names())
pos_weights = pos_nmf.components_

# extract and display topics and their components
pos_feature_names = np.array(ptvf.get_feature_names())
feature_idxs = np.argsort(-pos_weights)[:, :15]
topics = [pos_feature_names[idx] for idx in feature_idxs]
for idx, topic in enumerate(topics):
    print('Topic #'+str(idx+1)+':')
    print(', '.join(topic))
    print()
```

```
Topic #1:
but, one, make, no, take, way, even, get, seem, like, much, scene, may,
character, go

Topic #2:
movie, watch, see, like, think, really, good, but, see movie, great, movie
not, would, get, enjoy, say

Topic #3:
show, episode, series, tv, season, watch, dvd, television, first, good,
see, would, air, great, remember

Topic #4:
family, old, young, year, life, child, father, mother, son, year old, man,
friend, kid, boy, girl

Topic #5:
performance, role, actor, play, great, cast, good, well, excellent,
character, story, star, also, give, acting

Topic #6:
film, see, see film, film not, watch, good film, watch film, dvd, great
film, film but, film see, release, year, film make, great
```

```
Topic #7:
love, love movie, story, love story, fall love, fall, beautiful, song,
wonderful, music, heart, romantic, romance, favorite, character

Topic #8:
funny, laugh, hilarious, joke, humor, moment, fun, guy, get, but, line,
show, lot, time, scene

Topic #9:
ever, ever see, movie ever, one good, one, see, good, ever make, good
movie, movie, make, amazing, never, every, movie one

Topic #10:
comedy, romantic, laugh, hilarious, fun, humor, comic, joke, drama, light,
romance, star, british, classic, one
```

While some of the topics might be very generic, we can see that some of the topics clearly indicate the specific aspects from the reviews, which led to them having a positive sentiment. You can leverage pyLDAvis to visualize these topics in an interactive visualization.

```
In [14]: pyLDAvis.sklearn.prepare(pos_nmf, ptvf_features, ptvf, mds='mmds')
```

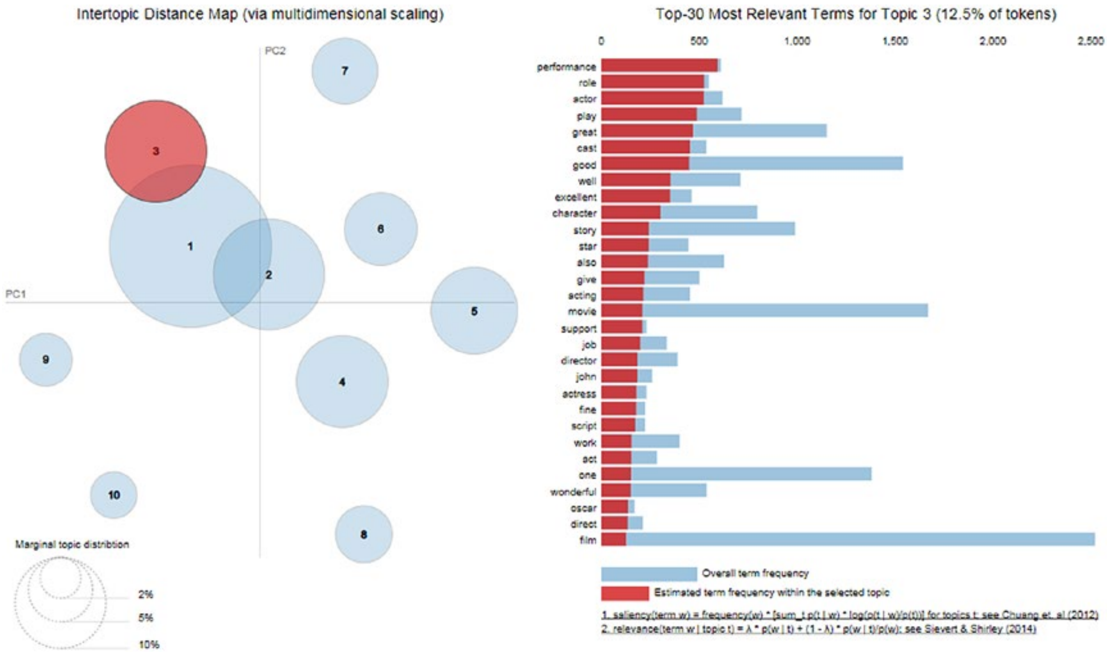


Figure 9-19. *Visualizing topic models on positive sentiment movie reviews*

The visualization in Figure 9-19 shows us the 10 topics from positive movie reviews and we can see the top relevant terms for Topic 5 from our previous output (pyLDAvis gives its own ordering to topics). From the topics and the terms, we can see terms like movie cast, actors, performance, play, characters, music, wonderful, script, good, and so on contribute to positive sentiment in various topics. This is quite interesting and gives you good insight into components of the reviews that contribute to the positive sentiment of the reviews. This visualization is completely interactive if you are using the Jupyter notebook. You can click on any of the bubbles representing topics in the Intertopic Distance Map on the left to see the most relevant terms in each of the topics in the bar chart on the right.

The plot on the left is rendered using multi-dimensional scaling (MDS). Similar topics should be close to one another and dissimilar topics should be far apart. The size of each topic bubble is based on the frequency of that topic and its components in the overall corpus.

The visualization on the right shows the top terms. When no topic it selected, it shows the top 30 most salient terms in the corpus. A term's *saliency* is defined as a measure of how frequently the term appears in the corpus and its distinguishing factor when used to distinguish between topics. When a topic is selected, the chart changes to show something similar to Figure 9-19, which shows the top 30 most relevant terms for that topic. The relevancy metric is controlled by λ, which can be changed based on a slider on top of the bar chart (refer to the notebook to interact with this). Readers interested in more mathematical theory behind these visualizations are encouraged to check out `https://cran.r-project.org/web/packages/LDAvis/vignettes/details.pdf`, which is a vignette for the R package LDAvis, which has been ported to Python as pyLDAvis.

Let's now extract topics and run this same analysis on our negative sentiment reviews from the movie reviews dataset.

```
# build topic model on negative sentiment review features
neg_nmf = NMF(n_components=total_topics, solver='cd', max_iter=500,
              random_state=42, alpha=.1, l1_ratio=.85)
neg_nmf.fit(ntvf_features)

# extract features and component weights
neg_feature_names = ntvf.get_feature_names()
neg_weights = neg_nmf.components_
```

```
# extract and display topics and their components
neg_feature_names = np.array(ntvf.get_feature_names())
feature_idxs = np.argsort(-neg_weights)[:, :15]
topics = [neg_feature_names[idx] for idx in feature_idxs]
for idx, topic in enumerate(topics):
    print('Topic #'+str(idx+1)+':')
    print(', '.join(topic))
    print()
```

```
Topic #1:
but, one, character, get, go, like, no, scene, seem, take, show, much,
time, would, play

Topic #2:
movie, watch, good, bad, think, like, but, see, would, make, even, movie
not, could, really, watch movie

Topic #3:
film, film not, good, bad, make, bad film, acting, film but, but, actor,
watch film, see film, script, watch, see

Topic #4:
horror, budget, low, low budget, horror movie, horror film, gore, flick,
zombie, blood, scary, killer, monster, kill, genre

Topic #5:
effect, special, special effect, fi, sci, sci fi, acting, bad, look, look
like, cheesy, terrible, cheap, creature, space

Topic #6:
funny, comedy, joke, laugh, not funny, show, humor, stupid, try, hilarious,
but, fun, suppose, episode, moment

Topic #7:
ever, ever see, bad, bad movie, movie ever, see, one bad, ever make, one,
movie, film ever, bad film, make, horrible, movie bad
```

```
Topic #8:
waste, waste time, time, not waste, money, complete, hour, spend, life,
talent, please, crap, total, plot, minute

Topic #9:
book, read, novel, story, version, base, character, change, write, love,
movie, comic, completely, miss, many

Topic #10:
year, old, year old, kid, child, year ago, ago, young, age, adult, boy,
girl, see, parent, school
```

While some of the topics might be very generic, just like we observed in the previous code segment, we can see some of the topics clearly indicate the specific aspects from the reviews which led to them having a negative sentiment. You can now leverage pyLDAvis to visualize these topics in an interactive visualization, just like the previous plot.

```
In [16]: pyLDAvis.sklearn.prepare(neg_nmf, ntvf_features, ntvf, mds='mmds')
```

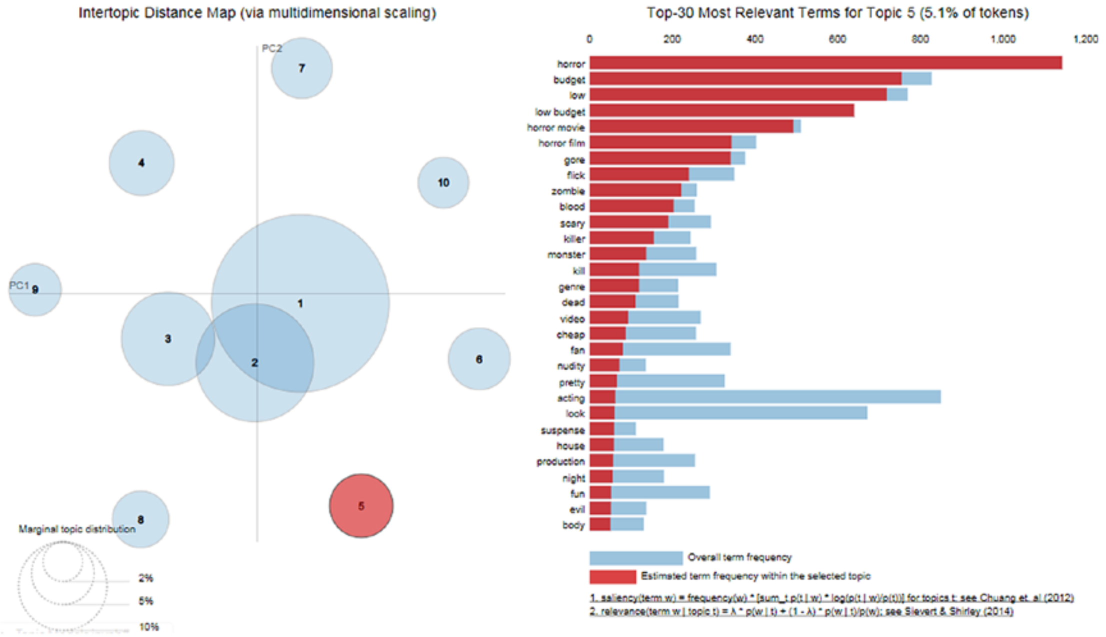


Figure 9-20. *Visualizing topic models on positive sentiment movie reviews*

The visualization in Figure 9-20 shows us the 10 topics from negative movie reviews and we can see the top relevant terms for Topic 4 highlighted in the output. From the topics and the terms, we can see terms like low budget, horror movie, gore, blood, cheap, scary, nudity, and so on have contributed to the negative sentiments. Of course, there are good chances of overlap between topics from positive and negative sentiment reviews but there will be distinguishable, distinct topics that further help us with interpretation and causal analysis.

Summary

This real-world case-study oriented chapter introduced the IMDB movie review dataset with the objective of predicting the sentiment of the reviews based on the textual content. We covered multiple aspects from NLP, including text preprocessing, normalization, feature engineering, and text classification. Unsupervised learning techniques using sentiment lexicons like TextBlob, Afinn, SentiWordNet, and Vader were covered in extensive detail, to show how we can analyze sentiment in the absence of labeled training data, which is a valid problem in today's organizations. Detailed workflow diagrams depicting text classification as a supervised machine learning problem helped us relate NLP to machine learning so that we can use machine learning techniques and methodologies to solve this problem of predicting sentiment when labeled data is available.

The focus on supervised methods was two-fold. This included traditional machine learning approaches and models like logistic regression and support vector machines and newer deep learning models including deep neural networks, RNNs, and LSTMs. Detailed concepts, workflows, hands-on examples, and comparative analyses with multiple supervised models and different feature engineering techniques have been covered for the purpose of predicting sentiment from movie reviews with maximum model performance. The final section of this chapter covered a very important aspect of machine learning, which is often neglected in our analyses. We looked at ways to analyze and interpret the cause of the positive or negative sentiments. Analyzing and visualizing model interpretations and topic models was covered with several examples, to give you good insight into how you can reuse these frameworks on your own datasets. The frameworks and methodologies used in this chapter should be useful for tackling similar problems in your own text data.

CHAPTER 10

The Promise of Deep Learning

The focus of this book has been primarily to get you up to speed on essential techniques in natural language processing, so covering detailed applications leveraging deep learning for NLP is out of the current scope. However, we have still tried to depict some interesting applications of NLP throughout the book, including Chapter 4, where we covered interesting methods around word embeddings using deep learning methods like Word2Vec, GloVe, and FastText and Chapter 5, where we built text classification models using deep learning. The intent of this chapter is to talk a fair bit about the recent advancements made in the field of NLP with the help of deep learning and the promise it holds toward building better models, solving more complex problems and helping us build better and more intelligent systems.

There has been a lot of hype with deep learning and artificial intelligence (AI) in general with skeptics portraying them as a failure and the media portraying a grim future with the loss of jobs and the rise of the so-called killer robots. The intent of this chapter is to cut through the hype and focus on the current reality of how these methods help build better and more generic systems with less effort on aspects like feature engineering and complex modeling. Deep learning has been delivering and is continuing to deliver continual success in areas like machine translation, text generation, text summarization, and speech recognition, which were all really tough problems to solve. Besides just solving them, they have also enabled us to reach human-level accuracy in the last couple of years!

Another interesting aspect is the scope of building universal models or representations such that we can represent any corpus of text in a vector space with minimal effort in feature engineering. The idea is to leverage some form of transfer learning such that we can use a pretrained model (which has been trained on huge

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1_10

corpora of rich textual data) to generalize representations on new text data, especially in problems with a lack of data. The best part of computer vision is that we have huge datasets like ImageNet with a suite of pretrained accurate models like VGG, Inception, and ResNet, which can be used as feature extractors in new problems. But what about NLP? Therein lies an inherent challenge considering that text data is so diverse, noisy, and unstructured. We've had some recent successes with word embeddings, including methods like Word2Vec, GloVe, and FastText, all of which are covered in Chapter 4. In this chapter, we will be showcasing several state-of-the-art generic sentence-embedding encoders, which tend to give surprisingly good performance especially on small amounts of data for transfer learning tasks as compared to word-embedding models. This will showcase the promise deep learning holds for NLP. We will be covering the following models:

- Averaged sentence embeddings
- Doc2Vec
- Neural-net language models (hands-on demo!)
- Skip-thought vectors
- Quick-thought vectors
- InferSent
- Universal sentence encoders

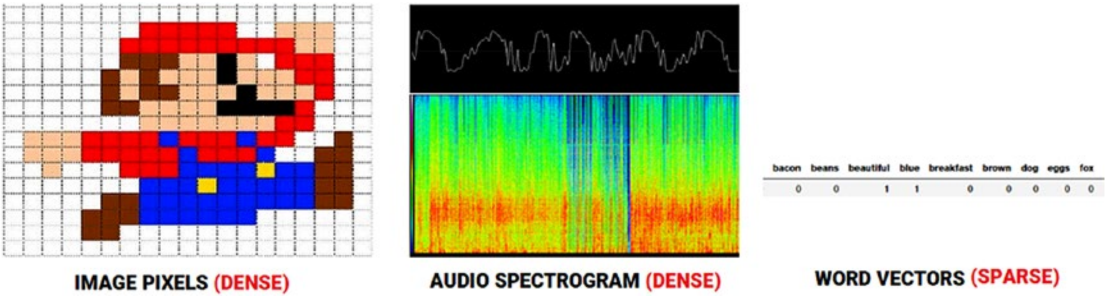
We cover the essential concepts and showcase some hands-on examples leveraging Python and TensorFlow in a text-classification problem focused on sentiment analysis based on the dataset from the previous chapter. This chapter is based on my recent thoughts, which I penned in a popular article that you can also access here if interested: https://towardsdatascience.com/deep-transfer-learning-for-natural-language-processing-text-classification-with-universal-1a2c69e5baa9. The intent of this chapter is not just to talk about some generic content around deep learning for NLP, but also to showcase cutting edge state-of-the-art deep transfer learning models for NLP with real world hands-on examples. Let's get started! All the code examples showcased in this chapter are available on the book's official GitHub repository, which you can access here: https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Why Are We Crazy for Embeddings?

What is this sudden craze behind embeddings? I'm sure many of you might be hearing it everywhere. Let's clear up the basics first and cut through the hype.

> *An embedding is a fixed-length vector typically used to encode and represent an entity (document, sentence, word, graph!).*

I've talked about the need for embeddings in the context of text data and NLP in Chapter 4 and in *one of my articles* at: https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa. But I will reiterate this briefly here for the sake of convenience. With regard to speech or image recognition systems, we already get information in the form of rich dense feature vectors embedded in high-dimensional datasets like audio spectrograms and image pixel intensities. However, when it comes to raw text data, especially count-based models like Bag of Words, we are dealing with individual words that may have their own identifiers and do not capture the semantic relationship among words. This leads to huge sparse word vectors for textual data and thus if we do not have enough data, we may end up getting poor models or even overfitting the data due to the curse of dimensionality. See Figure 10-1.



bacon	beans	beautiful	blue	breakfast	brown	dog	eggs	fox
0	0	1	1	0	0	0	0	0

Figure 10-1. *Comparing feature representations for audio, image, and text*

Predictive methods like *neural network based language models* try to predict words from neighboring words by looking at word sequences in the corpus. In the process, it learns distributed representations, thus giving us dense word embeddings.

Now you might be thinking, big deal, we get a bunch of vectors from text. What now? Well, if we have a good numeric representation of text data that captures even the context and semantics, we can use this for a wide variety of downstream real-world tasks like sentiment analysis, text classification, clustering, summarization, translation, and so on. The fact of the matter is, machine learning or deep learning models run on numbers and embeddings (see Figure 10-2) are they key to encoding text data to be used by these models.

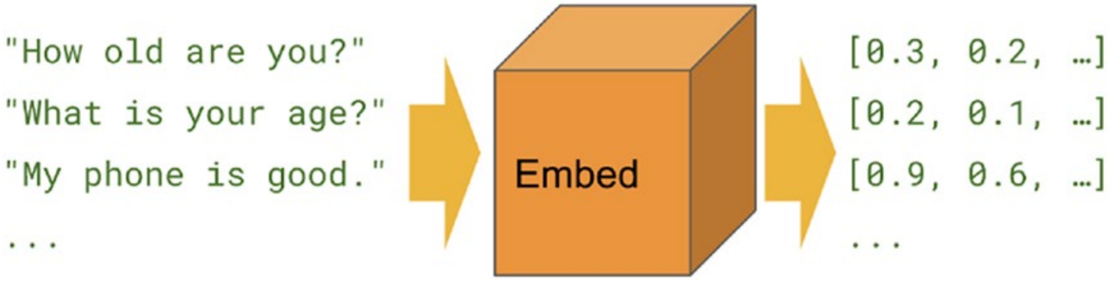
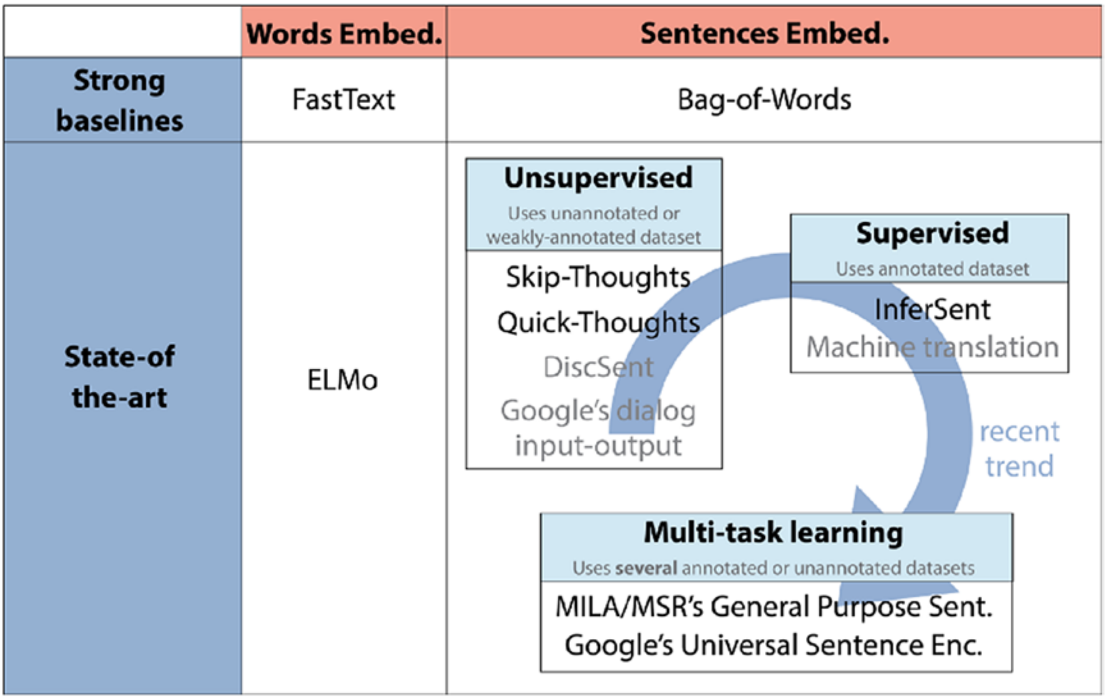


Figure 10-2. *Text embeddings*

A big trend here has been finding so-called "universal embeddings," which are basically pretrained embeddings obtained from training deep learning models on a huge corpus. This enables us to use these pretrained (generic) embeddings in a wide variety of tasks, including scenarios with constraints like lack of adequate data. This is a perfect example of transfer learning, in that it involves leveraging prior knowledge from pretrained embeddings to solve a completely new task! The following figure showcases some recent trends in universal word and sentence embeddings thanks to an amazing article (`https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a`) from the folks at HuggingFace (see Figure 10-3)!



	Words Embed.	Sentences Embed.
Strong baselines	FastText	Bag-of-Words
State-of the-art	ELMo	Unsupervised Uses unannotated or weakly-annotated dataset Skip-Thoughts Quick-Thoughts DiscSent Google's dialog input-output Supervised Uses annotated dataset InferSent Machine translation recent trend Multi-task learning Uses **several** annotated or unannotated datasets MILA/MSR's General Purpose Sent. Google's Universal Sentence Enc.

Figure 10-3. *Recent trends in universal word and sentence embeddings (Source: https://medium.com/huggingface/universal-word-sentence-embeddings-ce48ddc8fc3a)*

Figure 10-3 shows some interesting trends, including Google's universal sentence encoder, which we will be exploring in detail. Let's take a brief look at trends and developments in word- and sentence-embedding models before diving deeper into universal sentence encoder.

Trends in Word-Embedding Models

The word-embedding models are perhaps some of the older and more mature models that have been developed starting with Word2Vec in 2013. The three most common models leveraging deep learning (unsupervised approaches) based on embedding word vectors in a continuous vector space based on semantic and contextual similarity are:

- Word2Vec
- GloVe
- FastText

These models are based on the principle of *distributional hypothesis* in the field of *distributional semantics,* which tells us that words that occur and are used in the same context are semantically similar to one another and have similar meanings ("a word is characterized by the company it keeps"). Do refer to my article on word embeddings, as it covers these three methods in detail, if you are interested in the gory details!

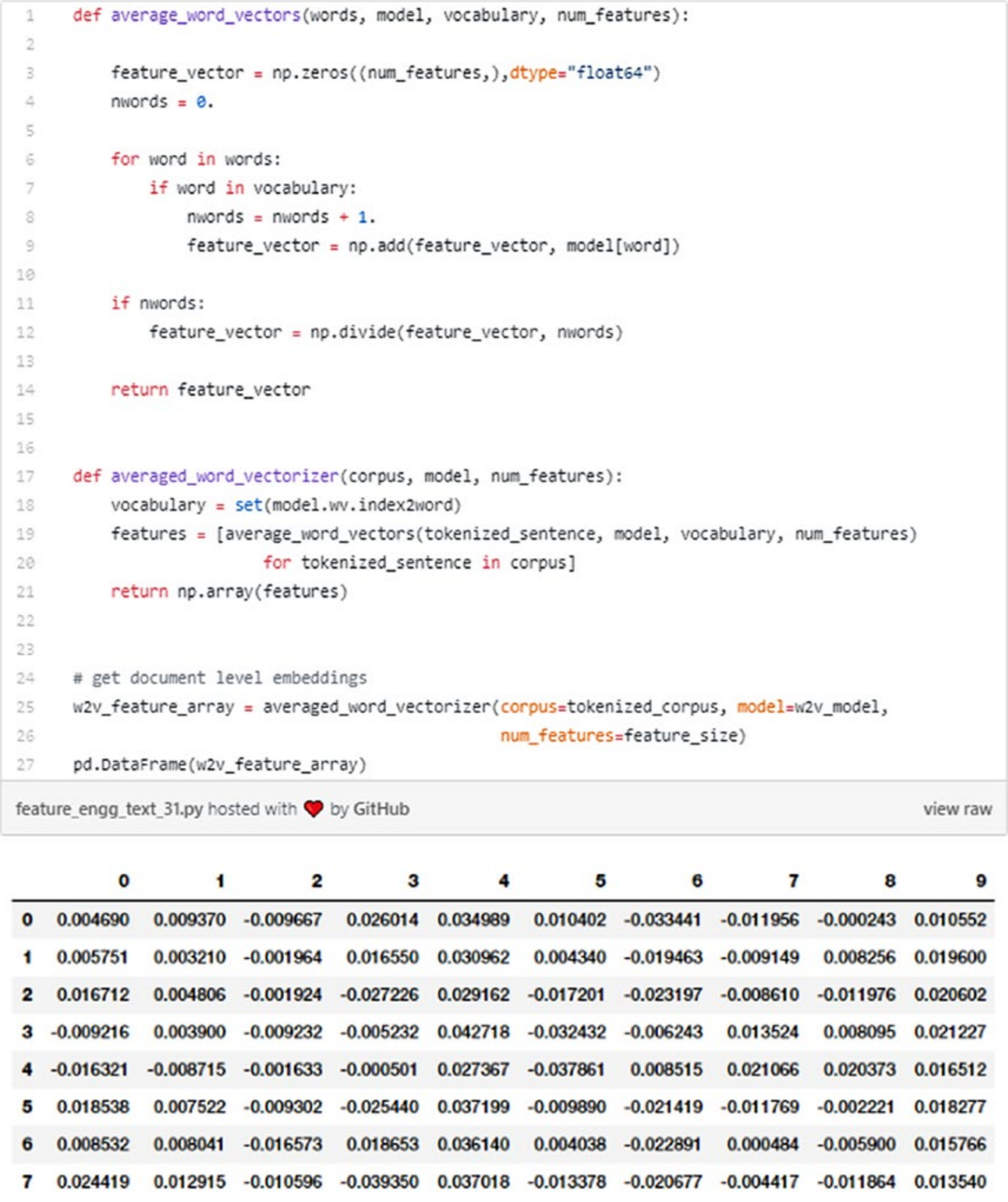
Another interesting model in this area that has been developed recently is *ELMo* (`https://allennlp.org/elmo`). It was developed by the Allen Institute for Artificial Intelligence. ELMo is a take on the famous Muppet character of the same name from the show "Sesame Street," but it's also an acronym for "Embeddings from Language Models".

ELMo gives us word embeddings that are learned from a deep bidirectional language model (biLM), which is typically pretrained on a large text corpus, enabling transfer learning and for these embeddings to be used across different NLP tasks. Allen AI tells us that ELMo representations are contextual, deep, and character-based. It uses morphological clues to form representations even for OOV (out-of-vocabulary) tokens.

Trends in Universal Sentence-Embedding Models

The concept of sentence embeddings is not new, because back when word embeddings were built, one of the easiest ways to build a baseline sentence-embedding model was with averaging.

A *baseline sentence-embedding model* can be built by averaging the individual word embeddings for every sentence/document (kind of similar to Bag of Words, where we lose that inherent context and sequence of words in the sentence). I do cover this in detail in my article (`https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa`), as well as in Chapter 5. Figure 10-4 shows a way of implementing this.



```python
def average_word_vectors(words, model, vocabulary, num_features):

    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])

    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector


def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.wv.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary, num_features)
                    for tokenized_sentence in corpus]
    return np.array(features)


# get document level embeddings
w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus, model=w2v_model,
                                              num_features=feature_size)
pd.DataFrame(w2v_feature_array)
```

feature_engg_text_31.py hosted with ❤ by GitHub   view raw

	0	1	2	3	4	5	6	7	8	9
0	0.004690	0.009370	-0.009667	0.026014	0.034989	0.010402	-0.033441	-0.011956	-0.000243	0.010552
1	0.005751	0.003210	-0.001964	0.016550	0.030962	0.004340	-0.019463	-0.009149	0.008256	0.019600
2	0.016712	0.004806	-0.001924	-0.027226	0.029162	-0.017201	-0.023197	-0.008610	-0.011976	0.020602
3	-0.009216	0.003900	-0.009232	-0.005232	0.042718	-0.032432	-0.006243	0.013524	0.008095	0.021227
4	-0.016321	-0.008715	-0.001633	-0.000501	0.027367	-0.037861	0.008515	0.021066	0.020373	0.016512
5	0.018538	0.007522	-0.009302	-0.025440	0.037199	-0.009890	-0.021419	-0.011769	-0.002221	0.018277
6	0.008532	0.008041	-0.016573	0.018653	0.036140	0.004038	-0.022891	0.000484	-0.005900	0.015766
7	0.024419	0.012915	-0.010596	-0.039350	0.037018	-0.013378	-0.020677	-0.004417	-0.011864	0.013540

***Figure 10-4.** Baseline sentence-embedding models*

Of course, there are more sophisticated approaches like encoding sentences in a linear weighted combination of their word embeddings and then remove some of the common principal components. Check out, "A Simple but Tough-to-Beat Baseline for Sentence Embeddings" at `https://openreview.net/forum?id=SyK00v5xx`.

Doc2Vec is also a very popular approach proposed by Mikolov et al. in their paper entitled "Distributed Representations of Sentences and Documents". Herein, they propose the paragraph vector, which is an unsupervised algorithm that learns fixed-length feature embeddings from variable-length pieces of text, such as sentences, paragraphs, and documents (see Figure 10-5).

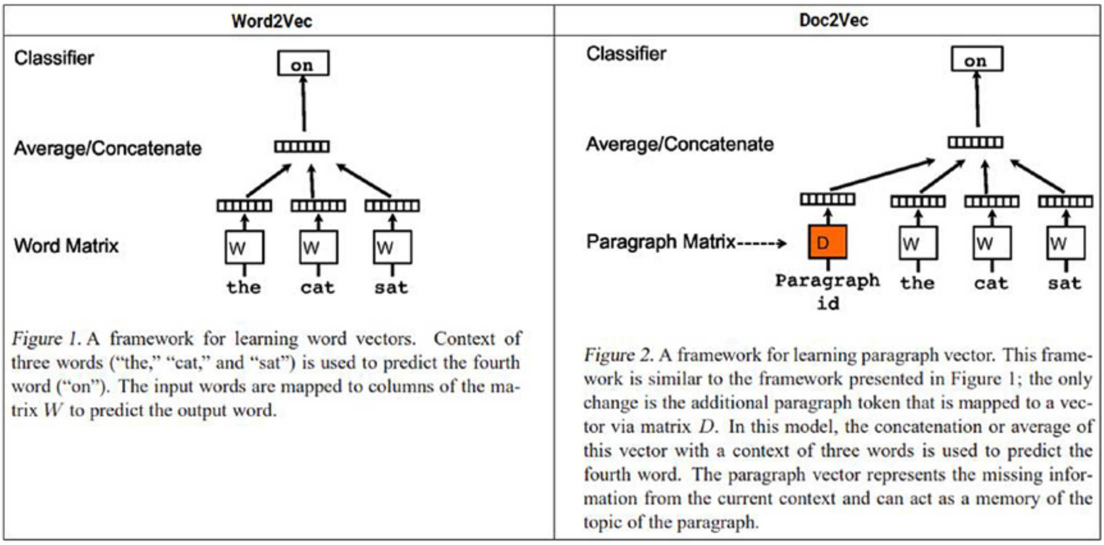


Figure 1. A framework for learning word vectors. Context of three words ("the," "cat," and "sat") is used to predict the fourth word ("on"). The input words are mapped to columns of the matrix W to predict the output word.

Figure 2. A framework for learning paragraph vector. This framework is similar to the framework presented in Figure 1; the only change is the additional paragraph token that is mapped to a vector via matrix D. In this model, the concatenation or average of this vector with a context of three words is used to predict the fourth word. The paragraph vector represents the missing information from the current context and can act as a memory of the topic of the paragraph.

Figure 10-5. *Word2Vec vs. Doc2Vec (source: `https://arxiv.org/abs/1405.4053`)*

Based on this depiction, the model represents each document by a dense vector, which is trained to predict words in the document. The only difference being the paragraph or document ID used along with the regular word tokens to build out the embeddings. Such a design, enables this model to overcome the weaknesses of bag-of-words models.

Neural-Net Language Models (NNLM) is a very early idea based on a neural probabilistic language model proposed by Bengio et al. in their 2013 paper, "A Neural Probabilistic Language Model," where they talk about learning a distributed representation for words that allows each training sentence to inform the model about an exponential number of semantically neighboring sentences. The model learns simultaneously a distributed representation for each word along with the probability function for word sequences, expressed in terms of these representations. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar (in the sense of having a nearby representation) to words forming an already seen sentence.

Google built a universal sentence-embedding model, `nnlm-en-dim128` (`https://tfhub.dev/google/nnlm-en-dim128/1`), which is a token-based text-embedding model trained using a three hidden layer feed-forward neural-net language model on the English Google News 200B corpus. This model maps any body of text into 128-dimensional embeddings. We will be using this in our hands-on demonstration shortly!

Skip-thought vectors were also one of the first models in the domain of unsupervised learning-based generic sentence encoders. In their proposed paper, "Skip-Thought Vectors," using the continuity of text from books, they trained an encoder-decoder model that tries to reconstruct the surrounding sentences of an encoded passage. Sentences that share semantic and syntactic properties are mapped to similar vector representations. See Figure 10-6.

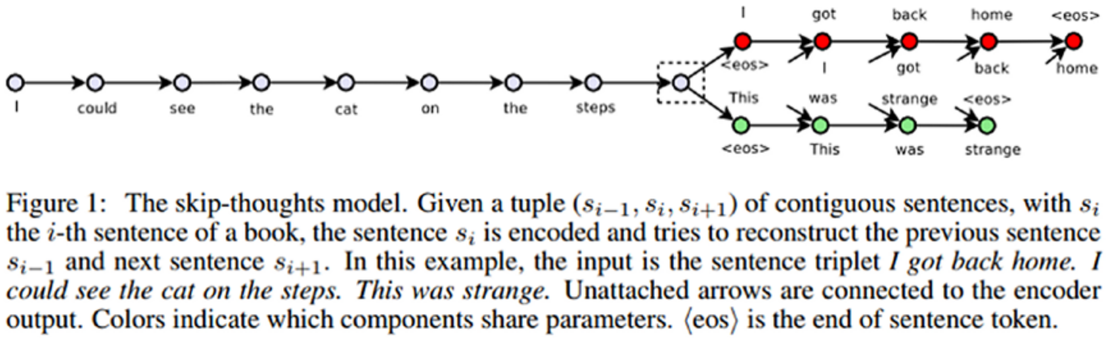


Figure 1: The skip-thoughts model. Given a tuple (s_{i-1}, s_i, s_{i+1}) of contiguous sentences, with s_i the i-th sentence of a book, the sentence s_i is encoded and tries to reconstruct the previous sentence s_{i-1} and next sentence s_{i+1}. In this example, the input is the sentence triplet *I got back home. I could see the cat on the steps. This was strange.* Unattached arrows are connected to the encoder output. Colors indicate which components share parameters. $\langle \text{eos} \rangle$ is the end of sentence token.

Figure 10-6. *Word2Vec vs. Doc2Vec (source:* `https://arxiv.org/abs/1405.4053`*)*

This is just like the Skip-gram model but for sentences, where we try to predict the surrounding sentences of a given source sentence.

Quick-thought vectors is a more recent unsupervised approach toward learning sentence embeddings (see Figure 10-7). Details are mentioned in the paper, "An efficient framework for learning sentence representations". Interestingly, they reformulate the problem of predicting the context in which a sentence appears as a classification problem by replacing the decoder with a classifier in the regular encoder-decoder architecture.

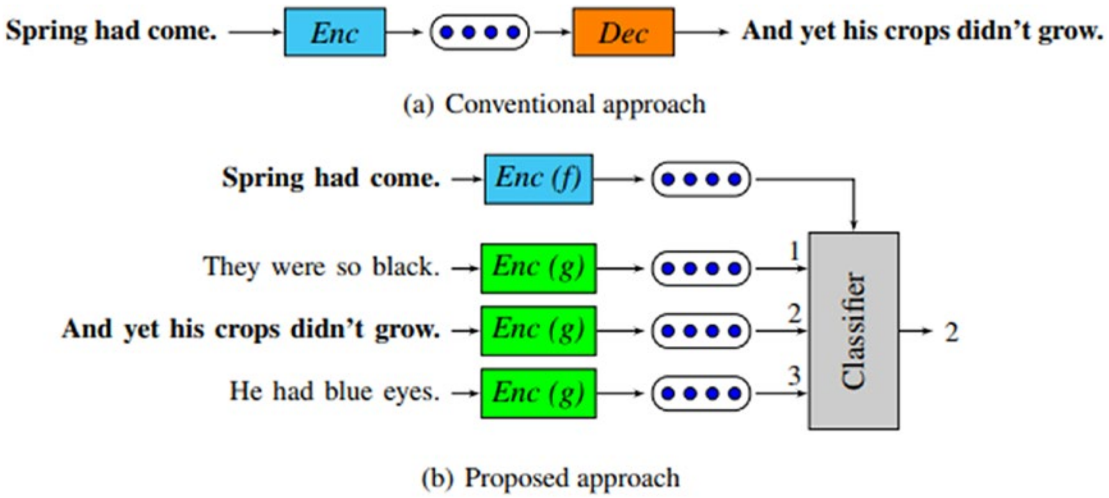


Figure 10-7. *Quick-thought vectors (source: `https://openreview.net/forum?id=rJvJXZb0W`)*

Thus, given a sentence and the context in which it appears, a classifier distinguishes context sentences from other contrastive sentences based on their embedding representations. An input sentence is first encoded by using some function. But instead of generating the target sentence, the model chooses the correct target sentence from a set of candidate sentences. Viewing generation as choosing a sentence from all possible sentences, this can be seen as a discriminative approximation to the generation problem.

InferSent is interestingly a supervised learning approach to learning universal sentence embeddings using natural language inference data. This is hardcore supervised transfer learning, where just like we get pretrained models trained on the ImageNet dataset for computer vision, they have universal sentence representations trained using supervised data from the Stanford natural language inference datasets. Details are mentioned in their paper, "Supervised Learning of Universal Sentence Representations from Natural Language Inference Data". The dataset used by this model is the SNLI dataset consists of 570,000 human-generated English sentence pairs, manually labeled with one of three categories: entailment, contradiction, or neutral. It captures natural language inference useful for understanding sentence semantics. See Figure 10-8.

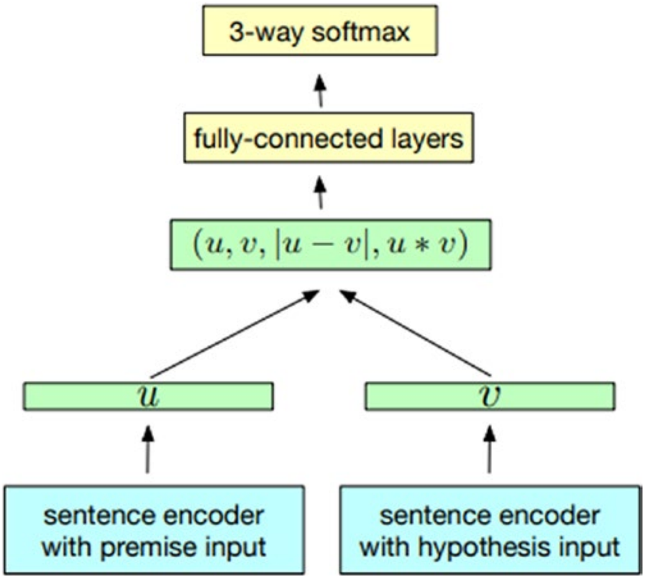

3-way softmax
fully-connected layers
$(u, v, |u - v|, u * v)$
u
v
sentence encoder with premise input
sentence encoder with hypothesis input

Figure 10-8. *InferSent training scheme (source: https://arxiv.org/abs/1705.02364)*

Based on the architecture depicted in Figure 10-8, we can see that it uses a shared sentence encoder that outputs a representation for the premise *u* and the hypothesis *v*. Once the sentence vectors are generated, three matching methods are applied to extract relations between *u* and *v*:

- Concatenation **(u, v)**
- Element-wise product **u** ∗ **v**
- Absolute element-wise difference **|u − v|**

The resulting vector is then fed into a three-class classifier consisting of multiple fully connected layers culminating in a softmax layer.

Universal Sentence Encoder, from Google, is one of the latest and best universal sentence-embedding models, and it was published in early 2018! The Universal Sentence Encoder encodes any body of text into 512-dimensional embeddings that can be used for a wide variety of NLP tasks, including text classification, semantic similarity, and clustering. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks that require modeling the meaning of sequences of words rather than just individual words.

Their key finding is that transfer learning using sentence embeddings tends to outperform word embedding level transfer. Check out their paper, "Universal Sentence Encoder" for further details. Essentially they have two versions of their model available in *TF-Hub* as universal-sentence-encoder (https://tfhub.dev/google/universal-sentence-encoder/2). Version 1 uses the transformer-network based sentence encoding model and Version 2 uses a Deep Averaging Network (DAN), where input embeddings for words and bi-grams are averaged together and then passed through a feed-forward deep neural network (DNN) to produce sentence embeddings. We will be using Version 2 in our hands-on demonstration shortly.

Understanding Our Text Classification Problem

It's time to put some of these universal sentence encoders into action with a hands-on demonstration! As the article mentions, the premise of our demonstration here is to focus on a very popular NLP task, text classification, in the context of sentiment analysis. We will be working with the benchmark IMDB Large Movie Review Dataset. Feel free to download it at http://ai.stanford.edu/~amaas/data/sentiment/ or you can even download it from my GitHub repository https://github.com/dipanjanS/data_science_for_all/tree/master/tds_deep_transfer_learning_nlp_classification. See Figure 10-9.

Figure 10-9. *Sentiment analysis on movie reviews*

This dataset consists of a total of 50,000 movie reviews, where 25,000 have positive sentiments and 25,000 have negative sentiments. We will be training our models on a total of 30,000 reviews as our training dataset, validate on 5,000 reviews, and use 15,000 reviews as our test dataset. The main objective is to correctly predict the sentiment of each review as positive or negative.

Universal Sentence Embeddings in Action

Now that we have defined our main objective, let's put universal sentence encoders into action! The code is available in the GitHub repository for this book at `https://github.com/dipanjanS/text-analytics-with-python`. Feel free to play around with it. I recommend using a GPU-based instance. I love using *Paperspace,* where you can spin up notebooks in the cloud without worrying about configuring instances manually.

My setup was an eight-CPU, 30GB, 250GB SSD and an NVIDIA Quadro P4000, which is usually cheaper than most AWS GPU instances (I love AWS though!).

Load Up Dependencies

We start by installing `tensorflow-hub`, which enables us to use these sentence encoders easily.

```
!pip install tensorflow-hub
Collecting tensorflow-hub
  Downloading https://files.pythonhosted.org/packages/5f/22/64f246ef80e64b
  1a13b2f463cefa44f397a51c49a303294f5f3d04ac39ac/tensorflow_hub-0.1.1-py2.
  py3-none-any.whl (52kB)
    100% |################################| 61kB 8.5MB/s ta 0:00:011
Requirement already satisfied: numpy>=1.12.0 in /usr/local/lib/python3.6/
dist-packages (from tensorflow-hub) (1.14.3)
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/
dist-packages (from tensorflow-hub) (1.11.0)
Requirement already satisfied: protobuf>=3.4.0 in /usr/local/lib/python3.6/
dist-packages (from tensorflow-hub) (3.5.2.post1)
Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-
packages (from protobuf>=3.4.0->tensorflow-hub) (39.1.0)
Installing collected packages: tensorflow-hub
Successfully installed tensorflow-hub-0.1.1
```

Let's now load our essential dependencies for this tutorial!

```
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import pandas as pd
```

The following commands help you check if `tensorflow` will be using a GPU (if you have one set up already):

```
In [12]: tf.test.is_gpu_available()
Out[12]: True
```

```
In [13]: tf.test.gpu_device_name()
Out[13]: '/device:GPU:0'
```

Load and View the Dataset

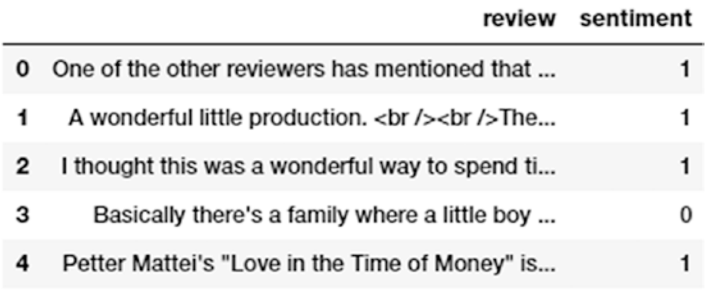
We can now load the dataset and view it using `pandas`. I provide a compressed version of the dataset in my repository, which you can use as follows.

```
dataset = pd.read_csv('movie_reviews.csv.bz2', compression='bz2')
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 2 columns):
review       50000 non-null object
sentiment    50000 non-null object
dtypes: object(2)
memory usage: 781.3+ KB
```

We encode the sentiment column as 1s and 0s just to make things easier during model development (label encoding). See Figure 10-10.

```
dataset['sentiment'] = [1 if sentiment == 'positive' else 0 for sentiment
in dataset['sentiment'].values]
dataset.head()
```



	review	sentiment
0	One of the other reviewers has mentioned that ...	1
1	A wonderful little production. <br /><br />The...	1
2	I thought this was a wonderful way to spend ti...	1
3	Basically there's a family where a little boy ...	0
4	Petter Mattei's "Love in the Time of Money" is...	1

Figure 10-10. *Our movie review dataset*

Building Train, Validation, and Test Datasets

We will now create the train, validation, and test datasets before we start modeling. We will use 30,000 reviews for the train dataset, 5,000 for the validation dataset, and 15,000 for the test dataset. You can use a `train-test` splitting function also, like `train_test_split()` from `scikit-learn`.

```
reviews = dataset['review'].values
sentiments = dataset['sentiment'].values

train_reviews = reviews[:30000]
train_sentiments = sentiments[:30000]

val_reviews = reviews[30000:35000]
val_sentiments = sentiments[30000:35000]

test_reviews = reviews[35000:]
test_sentiments = sentiments[35000:]
train_reviews.shape, val_reviews.shape, test_reviews.shape

((30000,), (5000,), (15000,))
```

Basic Text Wrangling

There is some basic text wrangling and preprocessing we need to do to remove some noise from our text, like the contractions, unnecessary special characters, HTML tags, and so on. The following code helps us build a simple, yet effective text-wrangling system. Install the following libraries if you don't have them. If you want you can also reuse the text-wrangling module we built in Chapter 3.

```
!pip install contractions
!pip install beautifulsoup4
```

The following functions help us build our text-wrangling system.

```
import contractions
from bs4 import BeautifulSoup
import unicodedata
import re

def strip_html_tags(text):
    soup = BeautifulSoup(text, "html.parser")
    [s.extract() for s in soup(['iframe', 'script'])]
    stripped_text = soup.get_text()
    stripped_text = re.sub(r'[\r|\n|\r\n]+', '\n', stripped_text)
    return stripped_text

def remove_accented_chars(text):
    text = unicodedata.normalize('NFKD', text).encode('ascii', 'ignore').
    decode('utf-8', 'ignore')
    return text

def expand_contractions(text):
    return contractions.fix(text)

def remove_special_characters(text, remove_digits=False):
    pattern = r'[^a-zA-ZO-9\s]' if not remove_digits else r'[^a-zA-Z\s]'
    text = re.sub(pattern, ", text)
    return text

def pre_process_document(document):

    # strip HTML
    document = strip_html_tags(document)

    # lower case
    document = document.lower()

    # remove extra newlines (often might be present in really noisy text)
    document = document.translate(document.maketrans("\n\t\r", "   "))
    # remove accented characters
    document = remove_accented_chars(document)

    # expand contractions
    document = expand_contractions(document)

    # remove special characters and\or digits
    # insert spaces between special characters to isolate them
    special_char_pattern = re.compile(r'([{.(-)!}])')
    document = special_char_pattern.sub(" \\1 ", document)
    document = remove_special_characters(document, remove_digits=True)

    # remove extra whitespace
    document = re.sub(' +', ' ', document)
    document = document.strip()

    return document

pre_process_corpus = np.vectorize(pre_process_document)
```

Let's now preprocess our datasets using the function we implemented above.

```
train_reviews = pre_process_corpus(train_reviews)
val_reviews = pre_process_corpus(val_reviews)
test_reviews = pre_process_corpus(test_reviews)
```

Build Data Ingestion Functions

Since we will be implementing our models in tensorflow using the tf.estimator API, we need to define some functions to build data and feature engineering pipelines to enable data flowing into our models during training. The following functions will help us. We leverage the numpy_input_fn() function, which feeds a dictionary of numpy arrays into the model.

```
# Training input on the whole training set with no limit on training epochs.
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'sentence': train_reviews}, train_sentiments,
    batch_size=256, num_epochs=None, shuffle=True)
```

```
# Prediction on the whole training set.
predict_train_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'sentence': train_reviews}, train_sentiments, shuffle=False)

# Prediction on the whole validation set.
predict_val_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'sentence': val_reviews}, val_sentiments, shuffle=False)

# Prediction on the test set.
predict_test_input_fn = tf.estimator.inputs.numpy_input_fn(
    {'sentence': test_reviews}, test_sentiments, shuffle=False)
```

We are now ready to build our models!

Build Deep Learning Model with Universal Sentence Encoder

We need to first define the sentence-embedding feature that leverages the Universal Sentence Encoder before building the model. We can do that using the following code.

```
embedding_feature = hub.text_embedding_column(
    key='sentence',
    module_spec="https://tfhub.dev/google/universal-sentence-encoder/2",
    trainable=False)
```

```
INFO:tensorflow:Using /tmp/tfhub_modules to cache modules.
```

Like we discussed, we use the Universal Sentence Encoder Version 2 and it works on the sentence attribute in our input dictionary, which will be a numpy array of our reviews. We will build a simple feed-forward DNN now with two hidden layers. Just a standard model—nothing too sophisticated since we want to see how well these embeddings perform even on a simple model. Here, we are leveraging transfer learning in the form of pretrained embeddings. We are not fine-tuning by keeping the embedding weights fixed by setting trainable=False.

```
dnn = tf.estimator.DNNClassifier(
          hidden_units=[512, 128],
          feature_columns=[embedding_feature],
          n_classes=2,
        activation_fn=tf.nn.relu,
        dropout=0.1,
        optimizer=tf.train.AdagradOptimizer(learning_rate=0.005))

# train for approx 12 epochs
# 256*1500 / 30000 == 12.8
```

We had set our batch_size to 256 and we will be flowing in data in batches of 256 records for 1,500 steps. This translates to roughly 12–13 epochs.

Model Training

Let's now train our model on our training dataset and evaluate on the train and validation datasets in steps of 100.

```
tf.logging.set_verbosity(tf.logging.ERROR)
import time

TOTAL_STEPS = 1500
STEP_SIZE = 100
for step in range(0, TOTAL_STEPS+1, STEP_SIZE):
    print()
    print('-'*100)
    print('Training for step =', step)
    start_time = time.time()
    dnn.train(input_fn=train_input_fn, steps=STEP_SIZE)
    elapsed_time = time.time() - start_time
    print('Train Time (s):', elapsed_time)
    print('Eval Metrics (Train):', dnn.evaluate(input_fn=predict_train_
    input_fn))
    print('Eval Metrics (Validation):', dnn.evaluate(input_fn=predict_val_
    input_fn))

--------------------------------------------------------------------
Training for step = 0
Train Time (s): 78.62789511680603
Eval Metrics (Train): {'accuracy': 0.84863335, 'accuracy_baseline':
0.5005, 'auc': 0.9279859, 'auc_precision_recall': 0.92819566, 'average_
loss': 0.34581015, 'label/mean': 0.5005, 'loss': 44.145977, 'precision':
0.86890674, 'prediction/mean': 0.47957155, 'recall': 0.8215118, 'global_
step': 100}
Eval Metrics (Validation): {'accuracy': 0.8454, 'accuracy_baseline': 0.505,
'auc': 0.92413086, 'auc_precision_recall': 0.9200026, 'average_loss':
0.35258815, 'label/mean': 0.495, 'loss': 44.073517, 'precision': 0.8522351,
'prediction/mean': 0.48447067, 'recall': 0.8319192, 'global_step': 100}

----------------------------------------------------------------------
Training for step = 100
Train Time (s): 76.1651611328125
Eval Metrics (Train): {'accuracy': 0.85436666, 'accuracy_baseline': 0.5005,
'auc': 0.9321357, 'auc_precision_recall': 0.93224275, 'average_loss':
0.3330773, 'label/mean': 0.5005, 'loss': 42.520508, 'precision': 0.8501513,
'prediction/mean': 0.5098621, 'recall': 0.86073923, 'global_step': 200}
Eval Metrics (Validation): {'accuracy': 0.8494, 'accuracy_baseline':
0.505, 'auc': 0.92772096, 'auc_precision_recall': 0.92323804, 'average_
loss': 0.34418356, 'label/mean': 0.495, 'loss': 43.022945, 'precision':
0.83501947, 'prediction/mean': 0.5149463, 'recall': 0.86707073, 'global_
step': 200}
----------------------------------------------------------------------
...
...
...
----------------------------------------------------------------------
Training for step = 1400
Train Time (s): 85.99037742614746
Eval Metrics (Train): {'accuracy': 0.8783, 'accuracy_baseline': 0.5005,
'auc': 0.9500882, 'auc_precision_recall': 0.94986326, 'average_loss':
0.28882334, 'label/mean': 0.5005, 'loss': 36.871063, 'precision': 0.865308,
'prediction/mean': 0.5196238, 'recall': 0.8963703, 'global_step': 1500}
Eval Metrics (Validation): {'accuracy': 0.8626, 'accuracy_baseline':
0.505, 'auc': 0.93708724, 'auc_precision_recall': 0.9336051, 'average_
loss': 0.32389137, 'label/mean': 0.495, 'loss': 40.486423, 'precision':
0.84044176, 'prediction/mean': 0.5226699, 'recall': 0.8917172, 'global_
step': 1500}
```

```
---------------------------------------------------------------------
Training for step = 1500
Train Time (s): 86.91469407081604
Eval Metrics (Train): {'accuracy': 0.8802, 'accuracy_baseline': 0.5005,
'auc': 0.95115364, 'auc_precision_recall': 0.950775, 'average_loss':
0.2844779, 'label/mean': 0.5005, 'loss': 36.316326, 'precision': 0.8735527,
'prediction/mean': 0.51057553, 'recall': 0.8893773, 'global_step': 1600}
Eval Metrics (Validation): {'accuracy': 0.8626, 'accuracy_baseline': 0.505,
'auc': 0.9373224, 'auc_precision_recall': 0.9336302, 'average_loss':
0.32108024, 'label/mean': 0.495, 'loss': 40.135033, 'precision': 0.8478599,
'prediction/mean': 0.5134171, 'recall': 0.88040406, 'global_step': 1600}
```

Based on the output logs, you can see that we get an overall accuracy of close to 87% on our validation dataset and an AUC of 94%, which is quite good on such a simple model!

Model Evaluation

Let's now evaluate our model and check the overall performance on the train and test datasets.

```
dnn.evaluate(input_fn=predict_train_input_fn)

{'accuracy': 0.8802, 'accuracy_baseline': 0.5005, 'auc': 0.95115364,
 'auc_precision_recall': 0.950775, 'average_loss': 0.2844779,
 'label/mean': 0.5005, 'loss': 36.316326, 'precision': 0.8735527,
 'prediction/mean': 0.51057553, 'recall': 0.8893773, 'global_step': 1600}
```

```
dnn.evaluate(input_fn=predict_test_input_fn)

{'accuracy': 0.8663333, 'accuracy_baseline': 0.5006667, 'auc': 0.9406502,
 'auc_precision_recall': 0.93988097, 'average_loss': 0.31214723, 'label/
 mean': 0.5006667,
 'loss': 39.679733, 'precision': 0.8597569, 'prediction/mean': 0.5120608,
 'recall': 0.8758988, 'global_step': 1600}
```

We get an overall accuracy of close to 87% on the test data, giving us consistent results based on what we observed on our validation dataset earlier. Thus, this should give you an idea of how easy it is to leverage pretrained universal sentence embeddings and not worry about the hassle of feature engineering or complex modeling.

Bonus: Transfer Learning with Different Universal Sentence Embeddings

Let's now try building different deep learning classifiers based on different sentence embeddings. We will try the following:

- NNLM-128
- USE-512

We will also cover the two most prominent methodologies for transfer learning here.

- Build a model using frozen pretrained sentence embeddings
- Build a model where we fine-tune and update the pretrained sentence embeddings during training

The following generic function can plug and play different universal sentence encoders from `tensorflow-hub`.

```
import time

TOTAL_STEPS = 1500
STEP_SIZE = 500

my_checkpointing_config = tf.estimator.RunConfig(
    keep_checkpoint_max = 2,       # Retain the 2 most recent checkpoints.
)

def train_and_evaluate_with_sentence_encoder(hub_module, train_
module=False, path=''):

    embedding_feature = hub.text_embedding_column(
        key='sentence', module_spec=hub_module, trainable=train_module)

    print()
    print('='*100)
print('Training with', hub_module)
print('Trainable is:', train_module)
print('='*100)

dnn = tf.estimator.DNNClassifier(
        hidden_units=[512, 128],
        feature_columns=[embedding_feature],
        n_classes=2,
        activation_fn=tf.nn.relu,
        dropout=0.1,
        optimizer=tf.train.AdagradOptimizer(learning_rate=0.005),
        model_dir=path,
        config=my_checkpointing_config)

for step in range(0, TOTAL_STEPS+1, STEP_SIZE):
    print('-'*100)
    print('Training for step =', step)
    start_time = time.time()
    dnn.train(input_fn=train_input_fn, steps=STEP_SIZE)
    elapsed_time = time.time() - start_time
    print('Train Time (s):', elapsed_time)
    print('Eval Metrics (Train):', dnn.evaluate(input_fn=predict_train_
    input_fn))
    print('Eval Metrics (Validation):', dnn.evaluate(input_fn=predict_
    val_input_fn))

train_eval_result = dnn.evaluate(input_fn=predict_train_input_fn)
test_eval_result = dnn.evaluate(input_fn=predict_test_input_fn)
return {
  "Model Dir": dnn.model_dir,
  "Training Accuracy": train_eval_result["accuracy"],
  "Test Accuracy": test_eval_result["accuracy"],
  "Training AUC": train_eval_result["auc"],
  "Test AUC": test_eval_result["auc"],
  "Training Precision": train_eval_result["precision"],
  "Test Precision": test_eval_result["precision"],
      "Training Recall": train_eval_result["recall"],
      "Test Recall": test_eval_result["recall"]
    }
```

We can now train our models using these defined approaches.

```
tf.logging.set_verbosity(tf.logging.ERROR)

results = {}

results["nnlm-en-dim128"] = train_and_evaluate_with_sentence_encoder(
    "https://tfhub.dev/google/nnlm-en-dim128/1", path='/storage/models/
    nnlm-en-dim128_f/')

results["nnlm-en-dim128-with-training"] = train_and_evaluate_with_sentence_
encoder(
    "https://tfhub.dev/google/nnlm-en-dim128/1", train_module=True, path='/
    storage/models/nnlm-en-dim128_t/')

results["use-512"] = train_and_evaluate_with_sentence_encoder(
    "https://tfhub.dev/google/universal-sentence-encoder/2", path='/
    storage/models/use-512_f/')

results["use-512-with-training"] = train_and_evaluate_with_sentence_encoder(
    "https://tfhub.dev/google/universal-sentence-encoder/2", train_
    module=True, path='/storage/models/use-512_t/')

====================================================================
Training with https://tfhub.dev/google/nnlm-en-dim128/1
Trainable is: False
====================================================================
--------------------------------------------------------------------
Training for step = 0
Train Time (s): 30.525171756744385
Eval Metrics (Train): {'accuracy': 0.8480667, 'auc': 0.9287864,
'precision': 0.8288572, 'recall': 0.8776557}
Eval Metrics (Validation): {'accuracy': 0.8288, 'auc': 0.91452694,
'precision': 0.7999259, 'recall': 0.8723232}
--------------------------------------------------------------------
```

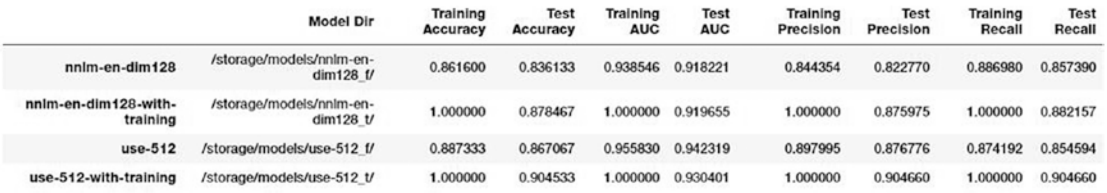
```
...
...
---------------------------------------------------------------------
Training for step = 1500
Train Time (s): 28.242169618606567
Eval Metrics (Train): {'accuracy': 0.8616, 'auc': 0.9385461, 'precision':
0.8443543, 'recall': 0.8869797}
Eval Metrics (Validation): {'accuracy': 0.828, 'auc': 0.91572505,
'precision': 0.80322945, 'recall': 0.86424243}
=====================================================================
Training with https://tfhub.dev/google/nnlm-en-dim128/1
Trainable is: True
=====================================================================
---------------------------------------------------------------------
Training for step = 0
Train Time (s): 45.97756814956665
Eval Metrics (Train): {'accuracy': 0.9997, 'auc': 0.9998141, 'precision':
0.99980015, 'recall': 0.9996004}
Eval Metrics (Validation): {'accuracy': 0.877, 'auc': 0.9225529,
'precision': 0.86671925, 'recall': 0.88808084}
---------------------------------------------------------------------
...
...
---------------------------------------------------------------------
Training for step = 1500
Train Time (s): 44.654765605926514
Eval Metrics (Train): {'accuracy': 1.0, 'auc': 1.0, 'precision': 1.0,
'recall': 1.0}
Eval Metrics (Validation): {'accuracy': 0.875, 'auc': 0.91479605,
'precision': 0.8661916, 'recall': 0.8840404}

=====================================================================
Training with https://tfhub.dev/google/universal-sentence-encoder/2
Trainable is: False
=====================================================================
---------------------------------------------------------------------
```

```
Training for step = 0
Train Time (s): 261.7671597003937
Eval Metrics (Train): {'accuracy': 0.8591, 'auc': 0.9373971, 'precision':
0.8820655, 'recall': 0.8293706}
Eval Metrics (Validation): {'accuracy': 0.8522, 'auc': 0.93081224,
'precision': 0.8631799, 'recall': 0.8335354}
--------------------------------------------------------------------
...
...
--------------------------------------------------------------------
Training for step = 1500
Train Time (s): 258.4421606063843
Eval Metrics (Train): {'accuracy': 0.88733333, 'auc': 0.9558296,
'precision': 0.8979955, 'recall': 0.8741925}
Eval Metrics (Validation): {'accuracy': 0.864, 'auc': 0.938815,
'precision': 0.864393, 'recall': 0.860202}
====================================================================
Training with https://tfhub.dev/google/universal-sentence-encoder/2
Trainable is: True
====================================================================
--------------------------------------------------------------------
Training for step = 0
Train Time (s): 313.1993100643158
Eval Metrics (Train): {'accuracy': 0.99916667, 'auc': 0.9996535,
'precision': 0.9989349, 'recall': 0.9994006}
Eval Metrics (Validation): {'accuracy': 0.9056, 'auc': 0.95068294,
'precision': 0.9020474, 'recall': 0.9078788}
--------------------------------------------------------------------
...
...
--------------------------------------------------------------------
Training for step = 1500
Train Time (s): 305.9913341999054
Eval Metrics (Train): {'accuracy': 1.0, 'auc': 1.0, 'precision': 1.0,
'recall': 1.0}
```

```
Eval Metrics (Validation): {'accuracy': 0.9032, 'auc': 0.929281,
'precision': 0.8986784, 'recall': 0.9066667}
```

I've depicted the evaluation metrics of importance in the output, and you can see we definitely get some good results with our models. The table in Figure 10-11 summarizes these comparative results in a nice way.

```
results_df = pd.DataFrame.from_dict(results, orient="index")
results_df
```



	Model Dir	Training Accuracy	Test Accuracy	Training AUC	Test AUC	Training Precision	Test Precision	Training Recall	Test Recall
nnlm-en-dim128	/storage/models/nnlm-en-dim128_f/	0.861600	0.836133	0.938546	0.918221	0.844354	0.822770	0.886980	0.857390
nnlm-en-dim128-with-training	/storage/models/nnlm-en-dim128_t/	1.000000	0.878467	1.000000	0.919655	1.000000	0.875975	1.000000	0.882157
use-512	/storage/models/use-512_f/	0.887333	0.867067	0.955830	0.942319	0.897995	0.876776	0.874192	0.854594
use-512-with-training	/storage/models/use-512_t/	1.000000	0.904533	1.000000	0.930401	1.000000	0.904660	1.000000	0.904660

Figure 10-11. *Comparing results from different universal sentence encoders*

Looks like Google's Universal Sentence Encoder with fine-tuning gave us the best results on the test data. Let's load this saved model and run an evaluation on the test data.

```
# get location of saved best model
best_model_dir = results_df[results_df['Test Accuracy'] == results_df['Test
Accuracy'].max()]['Model Dir'].values[0]

# load up model
embedding_feature = hub.text_embedding_column(
        key='sentence', module_spec="https://tfhub.dev/google/universal-
        sentence-encoder/2", trainable=True)

dnn = tf.estimator.DNNClassifier(
            hidden_units=[512, 128],
            feature_columns=[embedding_feature],
            n_classes=2,
            activation_fn=tf.nn.relu,
            dropout=0.1,
            optimizer=tf.train.AdagradOptimizer(learning_rate=0.005),
            model_dir=best_model_dir)
```

```
# define function to get model predictions
def get_predictions(estimator, input_fn):
    return [x["class_ids"][0] for x in estimator.predict(input_fn=input_fn)]

# get model predictions on test data
predictions = get_predictions(estimator=dnn, input_fn=predict_test_input_fn)
predictions[:10]
[0, 1, 0, 1, 1, 0, 1, 1, 1, 1]
```

One of the best ways to evaluate our model performance is to visualize the model predictions in the form of a confusion matrix (see Figure 10-12).

```
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

with tf.Session() as session:
    cm = tf.confusion_matrix(test_sentiments, predictions).eval()

LABELS = ['negative', 'positive']
sns.heatmap(cm, annot=True, xticklabels=LABELS, yticklabels=LABELS,
fmt='g')
xl = plt.xlabel("Predicted")
yl = plt.ylabel("Actuals")
```

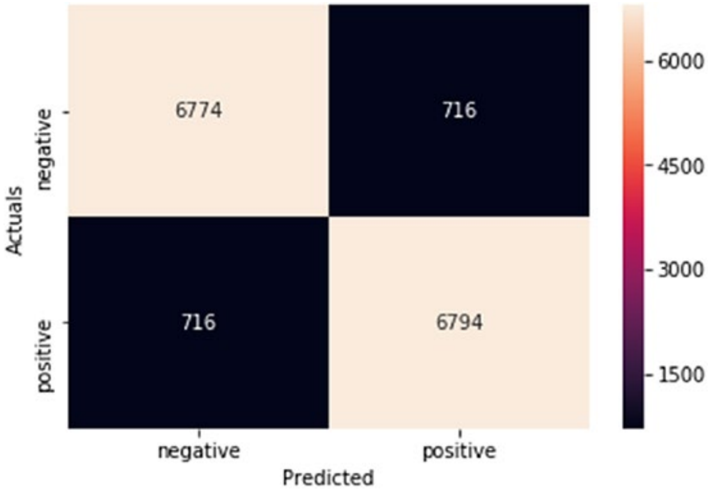


Figure 10-12. *Confusion matrix from our best model predictions*

We can also print out the model's classification report using Scikit-Learn to depict the other important metrics that can be derived from the confusion matrix, including precision, recall, and f1-score.

```
from sklearn.metrics import classification_report

print(classification_report(y_true=test_sentiments,
                            y_pred=predictions, target_names=LABELS))
```

```
             precision    recall  f1-score   support

   negative       0.90      0.90      0.90      7490
   positive       0.90      0.90      0.90      7510

avg / total       0.90      0.90      0.90     15000
```

We obtain an overall model accuracy and f1-score of 90% on the test data, which is really good. Go ahead and try this out. You might get an even better score; if so, let me know about it!

Summary and Future Scope

Universal sentence embeddings are definitely a huge step forward in enabling transfer learning for diverse NLP tasks. In fact, we have seen that models like ELMo, Universal Sentence Encoder, and ULMFiT have indeed made headlines by showcasing that pretrained models can be used to achieve state-of-the-art results on NLP tasks. I'm definitely excited about what the future holds for generalizing NLP even further and enabling us to solve complex tasks with ease!

This concludes the end of the last chapter in the book. I hope this enables you to go out there in the real world and apply some of the things you learned here to solve your own real-world problems in NLP. Always remember *Occam's Razor,* which states that the simplest solution is usually the best solution. While deep learning methods might be the cool thing right now, they are not the silver bullet for every solution. You should leverage them if and only if it makes perfect sense to do so, which you will better understand with intuition, experimentation, practicing, and reading over time. Now go out there and solve some interesting NLP problems and tell me about them!

Index

A

Adjective phrase (ADJP), 17, 18, 28, 165, 172
Advanced feature engineering models
  Bible corpus, load, 233
  FastText model, 269–272
  GloVe model, 263, 265, 267–269
  Word2Vec (*see* Word2Vec model)
Adverb phrase (ADVP), 18, 28, 165, 173, 358
Affinity propagation (AP) algorithm, 261, 499, 508–512
AFINN lexicon, 572, 578–580
American National Corpus (ANC), 54
Anaconda Python distribution, 55, 80–82
Application programming interfaces (APIs), 71, 73, 74, 285, 372, 521, 542
Artificial intelligence (AI), 62, 63, 67, 75, 112, 519, 521, 631
Artificial neural networks (ANNs), 68, 597
Association for Computational Linguistics (ACL), 569
Atomic units, 42
Automated document summarization, 346–347, 435
  abstraction-based techniques, 436
  extraction-based techniques, 436
  Gensim's summarization module, 438
  LSA
    low-rank SVD, 443
    salience score, 443–445
    SVD, 442
  Skyrim, 436, 437
  TextRank, 445–448, 450
  text wrangling, 439, 440
  TF-IDF feature engineering, 440, 441
  word count, 438

B

Backus-Naur Form (BNF), 89
Bagging, 306
Bag of N-Grams model, 210, 211, 295
Bag of Words model, 208–210, 220, 231, 234, 294, 315, 371, 419, 633
Baseline sentence-embedding model, 636–637
Batch normalization, 598
Beginning of sentence (BOS), 548
Behavioral theory, 7
Bing Liu's lexicon, 572, 574
Boosting, 307, 308
British National Corpus (BNC), 54, 142
Brown corpus, 53, 56
Bytes strings, 90

C

Canonical form, 36
Centroid-based clustering models, 499, 501
Centrum Wiskunde and Informatica (CWI), 70

D. Sarkar, *Text Analytics with Python*, https://doi.org/10.1007/978-1-4842-4354-1

Character vectorization, 459
Chart parsing, 192
Child language data exchange system (CHILDES), 53
Citation form, 36
Classification model
  bag of words, 315–319
  comparative model performance, 322, 323
  confusion matrix
    accuracy, 313
    logistic regression model, 310
    performance metrics, 312–315
    precision, 313
    recall, 314
    structure, 311, 312
  ensemble, 306, 307
  evaluation, 297
  FastText embeddings, 327, 328
  GBMs, 308, 309
  GloVe embeddings, 326
  logistic regression, 301–303
  metrics, 309
  model performance
    articles, 339, 340
    confusion matrix, 335–337
    consistency, 334, 335
    misclassified newsgroups, 337, 338
    prediction, 338, 339
    supervised learning algorithms, 340
    SVMs, 334
  model tuning, 329
    five-fold cross validation scheme, 329, 331, 332
    logistic regression, 332–334
    Naïve Bayes model, 329–331
    Scikit-Learn Pipeline object, 329
  multinomial Naïve Bayes, 298–301
  random forest, 307, 308
  SVMs, 303–306
  TF-IDF, 319–322
  training, 297
  tuning, 297
  Word2Vec, 323–325
Cleaning text, 288, 570
Cluster analysis, 497–500
Clustering algorithms, 457
Clustering movies
  AP algorithm, 508, 510–512
  feature engineering, 500, 501
  k-means clustering (*see* K-means clustering algorithm)
  Ward's agglomerative hierarchical clustering (*see* Ward's Agglomerative hierarchical clustering)
Collapsed Gibbs sampling, 391
Collins Birmingham University International Language Database (COBUILD), 53
Collins corpus, 53
Common language runtime (CLR), 77
Compositional semantics, 11
Computational Linguistics and Psycholinguistics (CLiPS), 575
Conditional random fields (CRF), 540, 547–550
Constant symbols, 47
Constituency grammar, 26, 52
  phrase structure rules, 27
  syntax trees
    NLTK and Stanford Core NLP, 32, 33
    noun phrases, 27, 28, 30, 31
    prepositional phrases, 29
    recursive properties, 29, 30

syntactic representation, 31
verb phrases, 28
Constructive dilemma, 45
Content-based classification, 278–279
Context free grammar (CFG), 32, 191, 195
Continuous Bag of Words (CBOW) model, 296
architecture, 234, 235
context word, 235, 237, 238
corpus vocabulary, build, 236, 237
deep learning architecture, 238–241
run, model, 241, 242
target word, 237–238
word embeddings, 242–244
Corpus of Contemporary American English (COCA), 54
Correcting spellings
case_of() function, 146, 147
known() function, 143–145
TextBlob library, 148
Cosine similarity, 487
bag of characters vectorization, 472
implementation, 474
term vectors, 471, 472
vector representations, 475
cossim() function, 487
CPython, 77

D

Database programming, 74
Data preparation, 162
Data preprocessing and normalization
normalize_corpus(...) function, 290, 291
simple pandas filter operation, 287
wrangling stage, 288, 289
Data retrieval, 285–287
Declarative clause, 20
Deep learning (DL), 68, 74, 631
classifiers
confusion matrix, 658
models training, 654–656
NNLM-128, 652
Scikit-Learn, 659
tensorflow-hub, 652–654
universal sentence encoders, 657, 658
USE-512, 652
Deep neural network (DNN), 68, 593, 597, 602, 642
Dendrogram, 224, 225, 498, 513–517
Density-based clustering models, 499
Dependency grammar, 22, 52
DAG, 23
syntactic function, 24
syntax tree, 23
tags, 25, 26
Directed acyclic graph (DAG), 23
Discourse analysis, 12
Disjunctive syllogism, 45
Distribution-based clustering models, 499
Doc2Vec
vs. Word2Vec model, 638, 639
Document, 347
Document clustering, 220, 222–226, 265, 280, 453–456, 497–500
Document frequencies (DF), 216, 217, 370, 395, 419, 440
Document similarity, 475
clustering
agglomerative hierarchical, 222, 223
cluster labels, 225, 226
dendrogram, 224, 225
linkage matrix, 223, 224

Document similarity (*cont.*)
  cosine similarity, 221, 222
  pairwise, 221
Document summarization, 280, 344, 346–347, 349, 362, 389, 447, 453

E

ElasticSearch, 497
Embedding, 231
  defined, 633
  predictive methods, 633
  text, 634
  universal sentence encoder, 635
Embeddings from language models (ELMo), 636
End of sentence (EOS), 548
Ensemble models, 306, 307, 309, 319, 322, 323
Escape sequences in strings, 90
Euclidean distance, 253, 462, 464, 465
Exclamative clause, 21
Execution speed performance, 75
Expanding contractions, 136–137, 206, 288, 570

F

Factual documents, 567
False negative (FN), 312
False positive (FP), 312, 314
FastText model, 296, 635
  Facebook, 269
  gensim.models.fasttext module, 270, 272
  subword model, 270
  vector representation, 270
Feature engineering, 293, 347, 455
  bag of N-grams model, 295
  bag of words model, 294
  embeddings, 295, 296
  label-encoding, 294
  one-hot encoding, 294
  TF-IDF model, 295
Feature extraction, 162
Feature matrix, 209, 210, 218, 220, 230, 348
First order logic (FOL), 560–562
  components, 47, 48
  definition, 46
  generalization, 50
  HOL, 51
  instantiation, 49, 50
  natural language statements, representation, 50, 51
  PL, 46, 48
  quantifier, 48–49
fit_transform(...) function, 220
Function symbols, 47

G

Gensim library, 113
Gensim framework, topic model
  feature engineering
    Bag of words model, 371
    document frequency, 370
    gensim.models.Phrases class, 369
    phrase generation model, 369
  LDA (*see* Latent Dirichlet allocation (LDA))
  LSI (*see* Latent semantic indexing (LSI))
  results
    distribution of dominant topics, 410, 411

dominant topics, research paper, 412
master dataframe, 410
predict topics, 415–418
relevant research paper, 413, 414
Global interpreter lock (GIL), 76
Global vectors (GloVe) model, 296, 635
matrix factorization, 264
SGD, 264
spaCy framework
en_vectors_web_lg model, 265
k-means clustering, 267, 268
vs. Word2Vec model, 268, 269
Google N-gram corpus, 54
Gradient boosting machines (GBMs), 308, 309
Graphical user interfaces (GUIs), 73
Groningen meaning bank (GMB), 545

H

Hadoop distributed file system (HDFS), 1
Hamming distance, 461, 462
Heterographs, 37
Heteronyms, 37
Hierarchical clustering models, 498, 512, 513
agglomerative, 513
dendrogram, 513–517
distance metric, 514
divisive, 513
linkage criterion, 514
Higher order logic (HOL), 51
Homophones, 36
Homonyms, 36, 524–525
Homographs, 36, 524–525
Human-computer interaction (HCI), 62
Hyperparameter, 284, 297
Hyponyms, 38, 527–528
Hypothetical syllogism, 45

I

Illocutionary acts, 8
ImageNet, 632
Imperative clause, 21
Indexing and slicing, 95–97
InferSent, 640, 641
Information overload, 344, 345
Information retrieval (IR), 64, 455
Inside, outside beginning (IOB) tagging, 547
Integrated development environments (IDEs), 79, 80
Internet movie database (IMDB), 568
Interrogative clause, 21
Inverse document frequency (IDF), 212, 217, 295
IronPython, 77

J

Java virtual machine (JVM), 77
Just in Time (JIT), 75, 77
Jython, 77

K

Keras, 238, 247
Kernel trick, 305
Keyphrase extraction, 344, 346
collocation
computation, 355–357
defined, 351
FreqDist class, 353
n-grams, 352, 353

Keyphrase extraction (*cont.*)
pointwise mutual information, 354
text_normalizer module, 351
terminology extraction, 350
uses, 350
weighted tag-based phrase extraction
chunker, 359
get_chunks() function, 360
POS tags, 357
text_normalizer module, 358
TextRank algorithm, 362
TF-IDF weights, 360
Keyword in context (KWIC), 53
K-means clustering algorithm
cluster movie data, 502–505
defined, 501
kmeans++ scheme, 502
NP hard problem, 501
pairwise document similarity, 506, 508
Kullback-Leibler (KL) divergence, 457

L

Label-encoding, 294
Lancaster-Oslo-Bergen (LOB) corpus, 53
Latent Dirichlet allocation (LDA), 227–229, 346, 363, 622
collapsed Gibbs sampling method, 391
core algorithm, 390
Gensim, 391
MALLET, 400–402
topic coherence, 393
aggregation, 396
confirmation measure, 395
four-stage pipeline, 394
probability calculation, 394
segmentation, 394
topic model, 391–393
coherence scores, 398, 399
output, 396, 397
tuning
best model, 405–407
coherence score, 403–405
optimal number of topics, 402, 403
term-topic dataframe, 407
topic-term dataframe, 408, 409
Latent semantic analysis (LSA), 232, 442–445
Latent semantic indexing (LSI), 227, 363
dense matrix, 382
document-topic matrix, 379
influential topics, sample papers, 387, 388
low-rank SVD, 382
model, 372–374
modelling themes, 375, 376, 378, 379
NIPS research papers, 372
SVD technique, 372
topics, sample papers, 380, 381
top influential terms, 383–387
Lemmatization, 152–154
Levenshtein edit distance
boundary conditions, 466
defined, 465
edit distance matrices, 471
implementation, 467–469
O(mn) space, 468
Python, 468
Lexical semantics, 11
Lexical similarity, 458
Lexicon, 11
Linguists, 10, 12
Lloyd’s algorithm, 501
Locutionary acts, 8

Logistic regression model, 301–303
Long short-term memory networks (LSTMs), 602, 603
Long strings, 90

M

Machine learning (ML), 67, 74
Machine learning for language toolkit (MALLET) framework, 399
Manhattan distance, 462–464
Maximum entropy (MaxEnt) classifier, 592
Maximum-likelihood estimation (MLE), 301
min_count parameter, 370
Model tuning, 298
Modus ponens, 44
Modus tollens, 45
Morphology, 11
Movie recommender
- components, 480
- cosine similarity, 482, 483
- matrix dataframe, input, 484
- Okapi BM25 (*see* Okapi BM25 ranking)
- popular movies, 485–487
- similar movies, 483, 484
- text preprocessing, 480, 481
- TF-IDF, 481, 482
- TMDB movies dataset, 477, 479, 480

Multi-class classification, 282
Multi-label classification, 282
Multi-layer perceptrons (MLPs), 597
Multinomial Naïve Bayes, 298–301
Multi-Perspective Question Answering (MPQA) subjectivity lexicon, 574, 575

N

Named entity recognition (NER)
- CRF sequence models, 540–542, 547–550
- end-to-end workflow, 554–557
- GMB, 545
- IOB tagging, 547
- JAR files, 540
- noun phrases, 536
- sklearn-crfsuite framework, 544, 551–554
- spaCy, 536, 537, 539
- Stanford's Core NLP, 542–544

Natural language
- cognition, 4
- cognitive learning, 7, 8
- definition, 3
- direction of fit, 5, 6
- language acquisition, 7, 8
- meaning of meaning model, 5
- relationship between language and reality, 4
- triangle of reference model, 4, 5
- usage, 4, 8–10

Natural language generation (NLG), 436
Natural language processing (NLP), 2, 12, 69, 75, 205, 567
- co-reference resolution, 64
- frameworks, 111–113
- HCI, 62
- machine translation, 62, 63
- QAS, 64
- speech recognition systems, 63
- text categorization, 65
- text summarization, 65
- word sense disambiguation, 64

Natural language toolkit (NLTK), 55, 56, 112
Negative polarity, 579
Negative sentiments, 568
Neural-Net language models (NNLM), 638
Neutral sentiments, 568
N-gram, 210
nltk library, 112
nltk.download() command, 56
nnlm-en-dim128, 639
Non-Negative matrix factorization (NNMF), 363, 428, 622–625
normalize_corpus(…) function, 572
Noun phrase (NP), 172
numpy_input_fn() function, 647

O

Object oriented principles (OOP), 70
Okapi BM25 ranking
  defined, 488
  Gensim framework, 489
  implementation, 489, 491–493
  mathematical definition, 488
  pairwise document, 493–495
  probabilistic relevancy, 488
  search engine, 497
  top five movie, 495–497
One-hot encoding, 294
One-vs-rest (OvR) scheme, 303
Operating systems (OSes), 79
Optical character recognition (OCR), 364
Out of vocabulary (OOV), 603

P

Package management, 84, 85
Padding (PAD) term, 237, 242
PageRank algorithm, 445
Partition-based or centroid-based clustering models, 499
Parts of speech (POS), 15, 52, 357, 572, 576
  ClassifierBasedPOSTagger class, 171, 172
  evaluate() function, 167
  feature_detector() function, 171
  NLTK, 166
  None tag, 170
  Penn treebank tags, 164, 165
  RegexpTagger, 168, 169
  sentence, 163
  SequentialBackoffTagger class, 169, 170
  spaCy, 166
  tag() function, 167
Pattern library, 112
Pattern lexicon, 575
Penn treebank, 54, 576
Perlocutionary acts, 8
Phonetics, 10
Phonology, 11
Positive sentiments, 568
Pragmatics, 12
Predicate symbols, 47
Predictive methods, 633
Prepositional phrase (PP), 173
Principal component analysis (PCA), 262
Probabilistic context-free grammar (PCFG), 193, 195, 199
Probabilistic statistical models, 346
Propositional logic (PL), 41, 46, 48, 558–560
  contradiction, 46
  inference, 44
  logical operators, 42
  logical proof, 44
  tautology, 46
  truth values, 43, 44

pyLDAvis framework, 434
PyPy, 77
Python, 2
  Anaconda Python distribution, 80–82
  applications, 73–75
  drawbacks, 75, 76
  IDEs, 79, 80
  implementations, 76–78
  Jupyter notebooks, 82, 83
  OSes, 79
  package management, 84, 85
  strings (*see* Text data)
  syntax and structure, 88, 89
  text data (*see* Text data)
  versions, 77–79
  virtual environment, 85–88
Python enhancement proposals (PEP) number 20 (PEP 20), 71
Python Package Index (PyPI), 73
Python standard library (PSL), 74

Q

Quantifiers, 47
Question answering systems (QAS), 64
Quick-thought vectors, 639, 640

R

Random forest, 307
Raw strings, 90
Recommender systems, 476, 477
re.compile() method, 103
Recurrent neural networks (RNNs), 602
Recursive descent parsing, 192
re.findall() method, 103
re.finditer() method, 103
Regular expressions (regexes), 102–106, 138
Regularization, 599
Reinforcement learning, 279
Relative clause, 21
re.match() method, 103
Removing accented characters, 288, 570
Removing special characters, 289, 571
Removing stopwords, 289, 571
replace() function, 142
Request-based classification, 278
re.search() method, 103
Resource description framework (RDF), 40
re.sub() method, 103
Reuters corpus, 54

S

Scientific computing, 74
Scikit-Learn framework, topic model
  Bag of Words model, 419
  LDA model
    maximum contribution, research papers, 426–428
    topic-term matrix, 426
  LSI model
    key topics, 424, 425
    SVD, 419
    topics generation, 420–423
  NMF model
    document-topic matrix, 430
    generated topics, 429
    LDA model, 431
    predict topics, 432–434
  pyLDAvis framework, 434
  visualizing, 434, 435
Scripting language, 73
Semantic analysis
  entailments, 524
  evaluate() function, 560

Semantic analysis (*cont.*)
FOL, 560
evaluation of expressions, 563–565
provers, 561
rule expression, 561–563
satisfiers() function, 565, 566
hypernym matrix, 531, 532
hyponyms and hypernyms, 526, 528, 529
NER (*see* Named entity recognition (NER))
PL, 558–560
relationships and similarity, 530–533
similarity matrix, 533
synonyms and antonyms, 525, 526
synsets, 522, 523
WordNet, 521
word sense disambiguation, 533–535
Semantics, 11
antonyms, 38
capitonyms, 37
definition, 35
FOL (*see* First order logic (FOL))
networks and models, 39, 40
OWL, 40
PL (*see* Propositional logic (PL))
polysemes, 37
RDF, 40
similarity, 458
synonyms, 37, 38
types and roles, 52
word forms, 36
Semiotics, 12
Semi-supervised learning, 279
Sentence tokenization
NLTK framework, 120
nltk.sent_tokenize(...) function, 121–123
pretrained tokenization model, 123–125
PunktSentenceTokenizer class, 125
RegexpTokenizer class, 125, 126
Sentiment analysis
IMDB, 568, 569
raw labeled movie reviews, 568, 569
text analytics, 569
Sentiment causation
model interpretation
classification model, 620–622
code, 616
negative review, 618
NNMF, 623–625
normalized text reviews, 616
normalized training reviews, 615, 616
positive review, 619, 620
root cause, 614
Scikit-Learn, 615
skater framework, 616, 617
TF-IDF, 622, 623
topic modeling, 622
visualization, 625–629
SentiWordNet lexicon, 580–583
Shift reduce parsing, 192
Short strings, 90
Sigmoid function, 591, 598
Similarity measures
distance metric of similarity, 456
KL-divergence, 457
Singular value decomposition (SVD), 346, 348–350
Skip-Gram model
architecture, 244, 245
context words, 245
implementation
corpus vocabulary, 246, 247
deep learning architecture, 248–251
generator, build, 247, 248
run, 251, 252

word embeddings
Euclidean distance metric, 253
t-SNE, 253, 254
word_model embedding layer, 252
Skip-thought vectors, 639
Skolemization, 50
Skyrim, 436
Slicing, 95–97
Social media, 343
Solr, 497
spacy library, 112
Stemming and lemmatization, 289, 571
Stochastic gradient descent (SGD), 264
Stopwords, 110, 111
String literals, 89–91
Stylistics, 12
Subjective documents, 567
Subject-verb-object (SVO) model, 22
Supervised deep learning models
batch normalization, 598
binary logistic regression function, 599
compile(...) method, 599
DNN, 597
document word vector averaging scheme, 595, 596
embedding layer, 606, 607
fit(...) function, 600
GloVe embeddings, 596
Keras, 600, 603
LSTMs, 602, 606, 608, 610–612
most_common(count) function, 604, 605
neural networks, 593
numeric representations, 594
one-hot encoded vectors, 593, 594
OOV, 603
regularization, 599
RNN, 602, 608, 609, 613
sequential model, 598
sigmoid function, 598
text classification system blueprint, 601, 602
validation dataset, 600, 601
word embeddings, 614
Word2Vec, 599, 601
Supervised learning, 171, 280, 281, 296–298, 303, 340, 453
Support vector machines (SVMs), 303–306
Syntax and structure
clauses, 20, 21
phrases, 17, 18, 20
POS, 15
SVO model, 22, 34, 35
word-order typology, 33–35
words, 15–17
Systems programming, 74

T

t-distributed stochastic neighbor embedding (t-SNE), 253, 254
tensorflow-hub, 643
Term frequencies (TF), 215, 295
Term frequency-inverse document frequency (TF-IDF) model, 295
DF, 216
IDF, 217
idf matrix, 218
mathematical representations, 212
TF, 215
tfidf matrix, 213, 219, 220
TfidfTransformer, 213, 214
TfidfVectorizer, 214, 215
Text analytics, 66, 67, 75
TextBlob lexicon, 575–577
Textblob library, 113, 148

Text classification, 642
- application, 341
- binary classification, 282
- blueprint, 282–285
- data retrieval, 285–287
- definition, 277, 278
- inherent properties, 275
- machine learning techniques, 279, 280
- prediction data, 282
- supervised machine learning algorithm, 276, 280
- task variants, 282
- textual data, 277
- train and test datasets, 292, 293
- training data, 281
- types, 278, 279
- unsupervised learning, 279

Text classifier
- classification model (*see* Classification model)

Text corpora
- brown corpus, 56–59
- monolingual corpora, 51
- NLTK, 55, 56
- nltk.download() command, 56
- popular resources, 53–55
- Reuters corpus, 59, 60
- treebanks, 52
- WordNet corpus, 60–62

Text corpus, 203–205
- accented characters/letters, 205
- contractions, 206
- lemmatization, 206
- special characters, 206
- stemming, 206
- stopwords, 206
- tags removal, 205
- text preprocessor, 207

Text correction
- repeating characters, 139–142
- wrong spellings (*see* Correcting spellings)

Text data, 89, 201–203
- string
  - basic operations, 94, 95
  - formatting, 100, 101
  - indexing and slicing, 95–97
  - literals, 89–91
  - methods, 98–100
  - regular expressions, 102–106
  - representation, 91–93

Text preprocessing and normalization, 570–572
- case conversions, 138, 139
- expanding contractions, 136, 137
- HTML tags, 117–119
- lemmatization, 152–154
- removing accented characters, 135
- sentence tokenization (*see* Sentence tokenization)
- special characters, 138
- stemming, 148–152
- stopwords, 154, 155
- text correction (*see* Text correction)
- text normalizer, 155–157
- word tokenization (*see* Word tokenization)

Text processing and analysis, 106–111

TextRank summarization algorithm, 439
- document similarity matrix, 447
- networkx library, 447, 448
- PageRank, 445, 446, 448, 449
- steps, 446

Text similarity, 454, 457, 458
- Cosine distance (*see* Cosine similarity)
- distance metrics, 460

Euclidean distance, 464, 465
Hamming distance, 461, 462
Levenshtein Edit distance (*see* Levenshtein Edit distance)
Manhattan distance, 462–464
NumPy arrays, 460
word representations, 458
Text syntax and structure
analytics and machine learning, 162
constituency parsing
algorithms, 191, 192
CFG, 191
NLTK and standord parser, 192, 193
nltk.CFG.fromstring() function, 194
PCFG, 193, 195, 199
phrase structure rules, 190, 191
treebank-based grammar, 196–198
dependency parsing
CLEAR dependency scheme, 186, 187
Graphviz, 187
NLP4J, 186
NLTK and the Stanford Dependency Parser, 187–190
root of sentence, 184
spaCy, 187
universal dependency, 184, 185
libraries and dependencies, 159–161
POS tagging (*see* Parts of speech (POS))
shallow parsing
chunking and chinking, 173–176
combined_taggers() function, 179
conll2000 data, 183
conll_tag_chunks() function, 179, 180
conlltags2tree()function, 178, 181
NGramTagChunker class, 180
parse() function, 180
phrases, 172, 173
treebank data, 181–182
tree2conlltags function, 178
techniques, 158
Text wrangling, 347
TF-IDF scheme, 475
Topic modeling, 344
Gensim (*see* Gensim framework, topic model)
matrix factorization, 363
research papers
data retrieval, 365
load and view dataset, 366, 367
NIPS conference, 364
objective, 364
text wrangling, 367, 368
Scikit-Learn framework (*see* Scikit-Learn framework, topic model)
SVD, 363
Traditional feature engineering models
Bag of N-Grams model, 210, 211
Bag of Words model, 208–210
document similarity (*see* Document similarity)
TF-IDF model (*see* Term frequency-inverse document frequency (TF-IDF) model)
topic modeling, 226
LDA, 227–231
LSI, 227
Traditional supervised machine learning models, 590–593
train-test splitting function, 645
transform(…) function, 220
treebanks, 52
True negative (TN), 312
True positive (TP), 312

U

Unicode, 89, 91
Universal embeddings, 634
Universal sentence encoders
    data ingestion, 647
    deep learning model, 648
    GitHub repository, 643
    Google, 641
    load and view data set, pandas, 644, 645
    load dependencies, 643, 644
    model evaluation, 651, 652
    model training, 649–651
    text wrangling, 645, 647
    TF-Hub, 642
Unsupervised learning algorithms, 457
Unsupervised lexicon-based models
    AFINN, 578–580
    Bing Liu's, 574
    MPQA subjectivity, 574, 575
    pattern, 575
    POS, 572
    SentiWordNet, 580–583
    TextBlob, 575–577
    VADER, 584–587

V

VADER lexicon, 584–587
Variable symbols, 47
Vector space models (VSMs), 203, 232
Verb phrase (VP), 172
Version incompatibility, 76
Virtual environment, 85–88

W, X, Y

Ward's agglomerative hierarchical clustering
    dendrogram, 514–516
    distance metric, 514
    linkage criterion, 514
Ward's minimum variance method, 223, 514
Web, chat, email, tweets, 55
Web development, 73
Web ontology language (OWL), 40
Web scraping, 118
Word-embedding models, 635, 636
Word lemmas, 52
WordNet, 54, 475
Word stems, 52
Word tokenization
    nltk.word_tokenize(...) function, 127
    RegexpTokenizer class, 129–132
    robust tokenizers, 132–135
    TokTokTokenizer, 128
    TreebankWordTokenizer, 127, 128
Word2Vec model, 295, 635
    CBOW model (*see* Continuous Bag of Words (CBOW) model)
    Gensim framework, 255
        implementation, 255
        model, 256, 257
        parameters, 255
        t-SNE, visualize, 257, 258
    machine learning tasks
        document-level embeddings, 260–263
        sample corpus, visualize, 258, 259
    Skip-Gram model (*see* Skip-Gram model)

Z

Zen of Python, 71, 72