

Krishna Bhavsar, Naresh Kumar,
Pratap Dangeti

Natural Language Processing with Python Cookbook

Over 60 recipes to implement text analytics solutions
using deep learning principles



Packt>

Natural Language Processing with Python Cookbook

Over 60 recipes to implement text analytics solutions using deep learning principles

Krishna Bhavsar
Naresh Kumar
Pratap Dangeti



BIRMINGHAM - MUMBAI

Natural Language Processing with Python Cookbook

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2017

Production reference: 1221117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-932-1

www.packtpub.com

Credits

Authors

Krishna Bhavsar
Naresh Kumar
Pratap Dangeti

Copy Editor

Vikrant Phadkay

Reviewer

Juan Tomas Oliva Ramos

Project Coordinator

Nidhi Joshi

Commissioning Editor

Veena Pagare

Proofreader

Safis Editing

Acquisition Editor

Aman Singh

Indexer

Tejal Daruwale Soni

Content Development Editor

Aishwarya Pandere

Graphics

Tania Dutta

Technical Editors

Dinesh Pawar
Suwarna Rajput

Production Coordinator

Shraddha Falebhai

About the Authors

Krishna Bhavsar has spent around 10 years working on natural language processing, social media analytics, and text mining in various industry domains such as hospitality, banking, healthcare, and more. He has worked on many different NLP libraries such as Stanford CoreNLP, IBM's SystemText and BigInsights, GATE, and NLTK to solve industry problems related to textual analysis. He has also worked on analyzing social media responses for popular television shows and popular retail brands and products. He has also published a paper on sentiment analysis augmentation techniques in 2010 NAACL. He recently created an NLP pipeline/toolset and open sourced it for public use. Apart from academics and technology, Krishna has a passion for motorcycles and football. In his free time, he likes to travel and explore. He has gone on pan-India road trips on his motorcycle and backpacking trips across most of the countries in South East Asia and Europe.

First and foremost, I would like to thank my mother for being the biggest motivating force and a rock-solid support system behind all my endeavors in life. I would like to thank the management team at Synerzip and all my friends for being supportive of me on this journey. Last but not least, special thanks to Ram and Dorothy for keeping me on track during this professionally difficult year.

Naresh Kumar has more than a decade of professional experience in designing, implementing, and running very-large-scale Internet applications in Fortune Top 500 companies. He is a full-stack architect with hands-on experience in domains such as e-commerce, web hosting, healthcare, big data and analytics, data streaming, advertising, and databases. He believes in open source and contributes to it actively. Naresh keeps himself up-to-date with emerging technologies, from Linux systems internals to frontend technologies. He studied in BITS-Pilani, Rajasthan with dual degree in computer science and economics.

Pratap Dangeti develops machine learning and deep learning solutions for structured, image, and text data at TCS, in its research and innovation lab in Bangalore. He has acquired a lot of experience in both analytics and data science. He received his master's degree from IIT Bombay in its industrial engineering and operations research program. Pratap is an artificial intelligence enthusiast. When not working, he likes to read about next-gen technologies and innovative methodologies. He is also the author of the book *Statistics for Machine Learning* by Packt.

I would like to thank my mom, Lakshmi, for her support throughout my career and in writing this book. I dedicate this book to her. I also thank my family and friends for their encouragement, without which it would not have been possible to write this book.

About the Reviewer

Juan Tomas Oliva Ramos is an environmental engineer from the University of Guanajuato, Mexico, with a master's degree in administrative engineering and quality. He has more than 5 years of experience in the management and development of patents, technological innovation projects, and the development of technological solutions through the statistical control of processes.

He has been a teacher of statistics, entrepreneurship, and the technological development of projects since 2011. He became an entrepreneur mentor and started a new department of technology management and entrepreneurship at Instituto Tecnológico Superior de Purísima del Rincón Guanajuato, Mexico.

Juan is an Alfaomega reviewer and has worked on the book *Wearable Designs for Smart Watches, Smart TVs and Android Mobile Devices*.

Juan has also developed prototypes through programming and automation technologies for the improvement of operations, which have been registered for patents.

I want to thank God for giving me wisdom and humility to review this book.

I thank Packt for giving me the opportunity to review this amazing book and to collaborate with a group of committed people

I want to thank my beautiful wife, Brenda, our two magic princesses (Maria Regina and Maria Renata) and our next member (Angel Tadeo), all of you, give me the strength, happiness, and joy to start a new day. Thanks for being my family.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details. At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/178728932X>. If you'd like to join our team of regular reviewers, you can email us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface	1
Chapter 1: Corpus and WordNet	8
Introduction	8
Accessing in-built corpora	9
How to do it...	9
Download an external corpus, load it, and access it	12
Getting ready	12
How to do it...	12
How it works...	14
Counting all the wh words in three different genres in the Brown corpus	15
Getting ready	15
How to do it...	15
How it works...	17
Explore frequency distribution operations on one of the web and chat text corpus files	17
Getting ready	18
How to do it...	18
How it works...	20
Take an ambiguous word and explore all its senses using WordNet	20
Getting ready	21
How to do it...	21
How it works...	24
Pick two distinct synsets and explore the concepts of hyponyms and hypernyms using WordNet	25
Getting ready	25
How to do it...	25
How it works...	28
Compute the average polysemy of nouns, verbs, adjectives, and adverbs according to WordNet	28
Getting ready	29
How to do it...	29
How it works...	30
Chapter 2: Raw Text, Sourcing, and Normalization	31
Introduction	31

The importance of string operations	32
Getting ready...	32
How to do it...	32
How it works...	34
Getting deeper with string operations	34
How to do it...	34
How it works...	37
Reading a PDF file in Python	37
Getting ready	37
How to do it...	38
How it works...	39
Reading Word documents in Python	40
Getting ready...	40
How to do it...	40
How it works...	43
Taking PDF, DOCX, and plain text files and creating a user-defined corpus from them	44
Getting ready	44
How to do it...	45
How it works...	47
Read contents from an RSS feed	48
Getting ready	48
How to do it...	48
How it works...	50
HTML parsing using BeautifulSoup	50
Getting ready	51
How to do it...	51
How it works...	53
Chapter 3: Pre-Processing	54
Introduction	54
Tokenization – learning to use the inbuilt tokenizers of NLTK	55
Getting ready	55
How to do it...	55
How it works...	57
Stemming – learning to use the inbuilt stemmers of NLTK	58
Getting ready	58
How to do it...	58
How it works...	60
Lemmatization – learning to use the WordnetLemmatizer of NLTK	60

Getting ready	60
How to do it...	61
How it works...	63
Stopwords – learning to use the stopwords corpus and seeing the difference it can make	63
Getting ready	63
How to do it...	63
How it works...	66
Edit distance – writing your own algorithm to find edit distance between two strings	66
Getting ready	66
How to do it...	67
How it works...	69
Processing two short stories and extracting the common vocabulary between two of them	69
Getting ready	69
How to do it...	70
How it works...	75
Chapter 4: Regular Expressions	76
Introduction	76
Regular expression – learning to use *, +, and ?	77
Getting ready	77
How to do it...	77
How it works...	79
Regular expression – learning to use \$ and ^, and the non-start and non-end of a word	79
Getting ready	80
How to do it...	80
How it works...	82
Searching multiple literal strings and substring occurrences	83
Getting ready	83
How to do it...	83
How it works...	85
Learning to create date regex and a set of characters or ranges of character	85
How to do it...	85
How it works...	87
Find all five-character words and make abbreviations in some sentences	88

How to do it...	88
How it works...	89
Learning to write your own regex tokenizer	89
Getting ready	89
How to do it...	90
How it works...	91
Learning to write your own regex stemmer	91
Getting ready	91
How to do it...	92
How it works...	93
Chapter 5: POS Tagging and Grammars	94
Introduction	94
Exploring the in-built tagger	95
Getting ready	95
How to do it...	95
How it works...	96
Writing your own tagger	97
Getting ready	97
How to do it...	98
How it works...	99
Training your own tagger	104
Getting ready	104
How to do it...	104
How it works...	106
Learning to write your own grammar	108
Getting ready	109
How to do it...	109
How it works...	110
Writing a probabilistic CFG	112
Getting ready	112
How to do it...	113
How it works...	114
Writing a recursive CFG	116
Getting ready	117
How to do it...	117
How it works...	119
Chapter 6: Chunking, Sentence Parse, and Dependencies	121
Introduction	121

Using the built-in chunker	121
Getting ready	122
How to do it...	122
How it works...	123
Writing your own simple chunker	124
Getting ready	126
How to do it...	126
How it works...	127
Training a chunker	129
Getting ready	130
How to do it...	130
How it works...	131
Parsing recursive descent	132
Getting ready	133
How to do it...	133
How it works...	135
Parsing shift-reduce	136
Getting ready	136
How to do it...	136
How it works...	138
Parsing dependency grammar and projective dependency	139
Getting ready	140
How to do it...	140
How it works...	141
Parsing a chart	142
Getting ready	143
How to do it...	143
How it works...	144
Chapter 7: Information Extraction and Text Classification	147
Introduction	147
Understanding named entities	148
Using inbuilt NERs	149
Getting ready	150
How to do it...	150
How it works...	152
Creating, inversing, and using dictionaries	153
Getting ready	153
How to do it...	153
How it works...	155

Choosing the feature set	159
Getting ready	159
How to do it...	159
How it works...	161
Segmenting sentences using classification	164
Getting ready	164
How to do it...	164
How it works...	166
Classifying documents	168
Getting ready	168
How to do it...	168
How it works...	170
Writing a POS tagger with context	173
Getting ready	173
How to do it...	173
How it works...	175
Chapter 8: Advanced NLP Recipes	178
Introduction	178
Creating an NLP pipeline	179
Getting ready	180
How to do it...	180
How it works...	182
Solving the text similarity problem	187
Getting ready	188
How to do it...	188
How it works...	190
Identifying topics	194
Getting ready	194
How to do it...	194
How it works...	196
Summarizing text	200
Getting ready	200
How to do it...	200
How it works...	202
Resolving anaphora	204
Getting ready	205
How to do it...	205
How it works...	207
Disambiguating word sense	210

Getting ready	211
How to do it...	211
How it works...	212
Performing sentiment analysis	213
Getting ready	214
How to do it...	214
How it works...	216
Exploring advanced sentiment analysis	218
Getting ready	218
How to do it...	218
How it works...	220
Creating a conversational assistant or chatbot	223
Getting ready	224
How to do it...	224
How it works...	227
Chapter 9: Applications of Deep Learning in NLP	230
Introduction	230
Convolutional neural networks	231
Applications of CNNs	235
Recurrent neural networks	235
Application of RNNs in NLP	237
Classification of emails using deep neural networks after generating TF-IDF	238
Getting ready	238
How to do it...	239
How it works...	240
IMDB sentiment classification using convolutional networks CNN 1D	247
Getting ready	247
How to do it...	249
How it works...	249
IMDB sentiment classification using bidirectional LSTM	252
Getting ready	253
How to do it...	253
How it works...	254
Visualization of high-dimensional words in 2D with neural word vector visualization	256
Getting ready	256
How to do it...	258
How it works...	258

Chapter 10: Advanced Applications of Deep Learning in NLP	263
Introduction	263
Automated text generation from Shakespeare's writings using LSTM	264
Getting ready...	264
How to do it...	265
How it works...	265
Questions and answers on episodic data using memory networks	270
Getting ready...	270
How to do it...	271
How it works...	272
Language modeling to predict the next best word using recurrent neural networks LSTM	278
Getting ready...	278
How to do it...	279
How it works...	280
Generative chatbot using recurrent neural networks (LSTM)	283
Getting ready...	283
How to do it...	284
How it works...	285
Index	289

Preface

Dear reader, thank you for choosing this book to pursue your interest in natural language processing. This book will give you a practical viewpoint to understand and implement NLP solutions from scratch. We will take you on a journey that will start with accessing inbuilt data sources and creating your own sources. And then you will be writing complex NLP solutions that will involve text normalization, preprocessing, POS tagging, parsing, and much more.

In this book, we will cover the various fundamentals necessary for applications of deep learning in natural language processing, and they are state-of-the-art techniques. We will discuss applications of deep learning using Keras software.

This book is motivated by the following goals:

- The content is designed to help newbies get up to speed with various fundamentals explained in a detailed way; and for experienced professionals, it will refresh various concepts to get more clarity when applying algorithms to chosen data
- There is an introduction to new trends in the applications of deep learning in NLP

What this book covers

Chapter 1, *Corpus and WordNet*, teaches you access to built-in corpora of NLTK and frequency distribution. We shall also learn what WordNet is and explore its features and usage.

Chapter 2, *Raw Text, Sourcing, and Normalization*, shows how to extract text from various formats of data sources. We will also learn to extract raw text from web sources. And finally we will normalize raw text from these heterogeneous sources and organize it in corpus.

Chapter 3, *Pre-Processing*, introduces some critical preprocessing steps, such as tokenization, stemming, lemmatization, and edit distance.

Chapter 4, *Regular Expressions*, covers one of the most basic and simple, yet most important and powerful, tools that you will ever learn. In this chapter, you will learn the concept of pattern matching as a way to do text analysis, and for this, there is no better tool than regular expressions.

Chapter 5, *POS Tagging and Grammars*. POS tagging forms the basis of any further syntactic analyses, and grammars can be formed and deformed using POS tags and chunks. We will learn to use and write our own POS taggers and grammars.

Chapter 6, *Chunking, Sentence Parse, and Dependencies*, helps you to learn how to use the inbuilt chunker as well as train/write your own chunker: dependency parser. In this chapter, you will learn to evaluate your own trained models.

Chapter 7, *Information Extraction and Text Classification*, tells you more about named entities recognition. We will be using inbuilt NEs and also creating your own named entities using dictionaries. Let's learn to use inbuilt text classification algorithms and simple recipes around its application.

Chapter 8, *Advanced NLP Recipes*, is about combining all your lessons so far and creating applicable recipes that can be easily plugged into any of your real-life application problems. We will write recipes such as text similarity, summarization, sentiment analysis, anaphora resolution, and so on.

Chapter 9, *Application of Deep Learning in NLP*, presents the various fundamentals necessary for working on deep learning with applications in NLP problems such as classification of emails, sentiment classification with CNN and LSTM, and finally visualizing high-dimensional words in low dimensional space.

Chapter 10, *Advanced Application of Deep Learning in NLP*, describes state-of-the-art problem solving using deep learning. This consists of automated text generation, question and answer on episodic data, language modeling to predict the next best word, and finally chatbot development using generative principles.

What you need for this book

To perform the recipes of this book successfully, you will need Python 3.x or higher running on any Windows- or Unix-based operating system with a processor of 2.0 GHz or higher and minimum 4 GB RAM. As far as the IDE for Python development are concerned, there are many available in the market but my personal favorite is PyCharm community edition. It's free, it's open source, and it's developed by JetBrains. That means support is excellent, advancement and fixes are distributed at a steady pace, and familiarity with IntelliJ keeps the learning curve pretty flat.

This book assumes you know Keras's basics and how to install the libraries. We do not expect that readers are already equipped with knowledge of deep learning and mathematics, such as linear algebra and so on.

We have used the following versions of software throughout this book, but it should run fine with any of the more recent ones also:

- Anaconda 3 – 4.3.1 (all Python and its relevant packages are included in Anaconda, Python – 3.6.1, NumPy – 1.12.1, pandas – 0.19.2)
- Theano – 0.9.0
- Keras – 2.0.2
- feedparser – 5.2.1
- bs4 – 4.6.0
- gensim – 3.0.1

Who this book is for

This book is intended for data scientists, data analysts, and data science professionals who want to upgrade their existing skills to implement advanced text analytics using NLP. Some basic knowledge of natural language processing is recommended.

This book is intended for any newbie with no knowledge of NLP or any experienced professional who would like to expand their knowledge from traditional NLP techniques to state-of-the-art deep learning techniques in the application of NLP.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it..., How it works..., There's more..., and See also). To give clear instructions on how to complete a recipe, we use these sections as follows.

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a new file named `reuters.py` and add the following import line in the file" A block of code is set as follows:

```
for w in reader.words(fileP):
    print(w + ' ', end='')
    if (w is '.'):
        print()
```

Any command-line input or output is written as follows:

```
# Deep Learning modules
>>> import numpy as np
>>> from keras.models import Sequential
```

New terms and important words are shown in bold.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account. Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Natural-Language-Processing-with-Python-Cookbook>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Corpus and WordNet

In this chapter, we will cover the following recipes:

- Accessing in-built corpora
- Download an external corpus, load it, and access it
- Counting all the wh words in three different genres in the Brown corpus
- Explore frequency distribution operations on one of the web and chat text corpus files
- Take an ambiguous word and explore all its senses using WordNet
- Pick two distinct synsets and explore the concepts of hyponyms and hypernyms using WordNet
- Compute the average polysemy of nouns, verbs, adjectives, and adverbs according to WordNet

Introduction

To solve any real-world **Natural Language Processing (NLP)** problems, you need to work with huge amounts of data. This data is generally available in the form of a corpus out there in the open diaspora and as an add-on of the NLTK package. For example, if you want to create a spell checker, you need a huge corpus of words to match against.

The goal of this chapter is to cover the following:

- Introducing various useful textual corpora available with NLTK
- How to access these in-built corpora from Python
- Working with frequency distributions
- An introduction to WordNet and its lexical features

We will try to understand these things from a practical standpoint. We will perform some exercises that will fulfill all of these goals through our recipes.

Accessing in-built corpora

As already explained, we have many corpora available for use with NLTK. We will assume that you have already downloaded and installed NLTK data on your computer. If not, you can find the same at <http://www.nltk.org/data.html>. Also, a complete list of corpora that you can use from within NLTK data is available at http://www.nltk.org/nltk_data/.

Now, our first task/recipe involves us learning how to access any one of these corpora. We have decided to do some tests on the Reuters corpus or the same. We will import the corpus into our program and try to access it in different ways.

How to do it...

1. Create a new file named `reuters.py` and add the following import line in the file. This will specifically allow access to only the `reuters` corpus in our program from the entire NLTK data:

```
from nltk.corpus import reuters
```

2. Now we want to check what exactly is available in this corpus. The simplest way to do this is to call the `fileids()` function on the corpus object. Add the following line in your program:

```
files = reuters.fileids()
print(files)
```

3. Now run the program and you shall get an output similar to this:

```
['test/14826', 'test/14828', 'test/14829', 'test/14832',
'test/14833', 'test/14839',
```

These are the lists of files and the relative paths of each of them in the `reuters` corpus.

4. Now we will access the actual content of any of these files. To do this, we will use the `words()` function on the corpus object as follows, and we will access the `test/16097` file:

```
words16097 = reuters.words(['test/16097'])
print(words16097)
```

5. Run the program again and an extra new line of output will appear:

```
['UGANDA', 'PULLS', 'OUT', 'OF', 'COFFEE', 'MARKET', ...]
```

As you can see, the list of words in the `test/16097` file is shown. This is curtailed though the entire list of words is loaded in the memory object.

6. Now we want to access a specific number of words (20) from the same file, `test/16097`. Yes! We can specify how many words we want to access and store them in a list for use. Append the following two lines in the code:

```
words20 = reuters.words(['test/16097'])[:20]
print(words20)
```

Run this code and another extra line of output will be appended, which will look like this:

```
['UGANDA', 'PULLS', 'OUT', 'OF', 'COFFEE', 'MARKET', '-', 'TRADE',
'SOURCES', 'Uganda', '"', 's', 'Coffee', 'Marketing', 'Board', '(',
'CMB', ')', 'has', 'stopped']
```

7. Moving forward, the `reuters` corpus is not just a list of files but is also hierarchically categorized into 90 topics. Each topic has many files associated with it. What this means is that, when you access any one of the topics, you are actually accessing the set of all files associated with that topic. Let's first output the list of topics by adding the following code:

```
reutersGenres = reuters.categories()
print(reutersGenres)
```

Run the code and the following line of output will be added to the output console:

```
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa',
'coconut', 'coconut-oil', ...]
```

All 90 categories are displayed.

8. Finally, we will write four simple lines of code that will not only access two topics but also print out the words in a loosely sentenced fashion as one sentence per line. Add the following code to the Python file:

```
for w in reuters.words(categories=['bop','cocoa']):
    print(w+' ',end='')
    if(w is '.'):
        print()
```

9. To explain briefly, we first selected the categories 'bop' and 'cocoa' and printed every word from these two categories' files. Every time we encountered a dot (.), we inserted a new line. Run the code and something similar to the following will be the output on the console:

```
['test/14826', 'test/14828', 'test/14829', 'test/14832',
'test/14833', 'test/14839', ...
['UGANDA', 'PULLS', 'OUT', 'OF', 'COFFEE', 'MARKET', ...]
['UGANDA', 'PULLS', 'OUT', 'OF', 'COFFEE', 'MARKET', '-', 'TRADE',
'SOURCES', 'Uganda', '"', 's', 'Coffee', 'Marketing', 'Board', '(',
'CMB', ')', 'has', 'stopped']
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa',
'coconut', 'coconut-oil', ...
SOUTH KOREA MOVES TO SLOW GROWTH OF TRADE SURPLUS South Korea ' s
trade surplus is growing too fast and the government has started
taking steps to slow it down , Deputy Prime Minister Kim Mahn-je
said .
He said at a press conference that the government planned to
increase investment , speed up the opening of the local market to
foreign imports, and gradually adjust its currency to hold the
surplus " at a proper level ." But he said the government would not
allow the won to appreciate too much in a short period of time .
South Korea has been under pressure from Washington to revalue the
won .
The U .
S .
Wants South Korea to cut its trade surplus with the U .
S ., Which rose to 7 .
4 billion dlrs in 1986 from 4 .
3 billion dlrs in 1985 .
.
.
.
```

Download an external corpus, load it, and access it

Now that we have learned how to load and access an inbuilt corpus, we will learn how to download and also how to load and access any external corpus. Many of these inbuilt corpora are very good use cases for training purposes, but for solving any real-world problem, you will normally need an external dataset. For this recipe's purpose, we will be using the **Cornell CS Movie** review corpus, which is already labelled for positive and negative reviews and used widely for training sentiment analysis modules.

Getting ready

First and foremost, you will need to download the dataset from the Internet. Here's the link: http://www.cs.cornell.edu/people/pabo/movie-review-data/mix20_rand700_tokens_cleaned.zip). Download the dataset, unzip it, and store the resultant `Reviews` directory at a secure location on your computer.

How to do it...

1. Create a new file named `external_corpus.py` and add the following import line to it:

```
from nltk.corpus import CategorizedPlaintextCorpusReader
```

Since the corpus that we have downloaded is already categorized, we will use `CategorizedPlaintextCorpusReader` to read and load the given corpus. This way, we can be sure that the categories of the corpus are captured, in this case, positive and negative.

2. Now we will read the corpus. We need to know the absolute path of the `Reviews` folder that we unzipped from the downloaded file from Cornell. Add the following four lines of code:

```
reader = CategorizedPlaintextCorpusReader(r'/Volumes/Data/NLP-  
CookBook/Reviews/txt_sentoken', r'.*\..txt', cat_pattern=r'(\w+)/*')  
print(reader.categories())  
print(reader.fileids())
```

The first line is where you are reading the corpus by calling the `CategorizedPlaintextCorpusReader` constructor. The three arguments from left to right are Absolute Path to the `txt_sentoken` folder on your computer, all sample document names from the `txt_sentoken` folder, and the categories in the given corpus (in our case, 'pos' and 'neg'). If you look closely, you'll see that all the three arguments are regular expression patterns. The next two lines will validate whether the corpus is loaded correctly or not, printing the associated categories and filenames of the corpus. Run the program and you should see something similar to the following:

```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt',  
'bible-kjv.txt', ...]  
[['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday',  
'an', 'investigation', 'of', ...]]
```

3. Now that we've made sure that the corpus is loaded correctly, let's get on with accessing any one of the sample documents from both the categories. For that, let's first create a list, each containing samples of both the categories, 'pos' and 'neg', respectively. Add the following two lines of code:

```
posFiles = reader.fileids(categories='pos')  
negFiles = reader.fileids(categories='neg')
```

The `reader.fileids()` method takes the argument category name. As you can see, what we are trying to do in the preceding two lines of code is straightforward and intuitive.

4. Now let's select a file randomly from each of the lists of `posFiles` and `negFiles`. To do so, we will need the `randint()` function from the `random` library of Python. Add the following lines of code and we shall elaborate what exactly we did immediately after:

```
from random import randint  
fileP = posFiles[randint(0, len(posFiles)-1)]  
fileN = negFiles[randint(0, len(posFiles) - 1)]  
print(fileP)  
print(fileN)
```

The first line imports the `randint()` function from the `random` library. The next two files select a random file, each from the set of positive and negative category reviews. The last two lines just print the filenames.

5. Now that we have selected the two files, let's access them and print them on the console sentence by sentence. We will use the same methodology that we used in the first recipe to print a line-by-line output. Append the following lines of code:

```
for w in reader.words(fileP):
    print(w + ' ', end='')
    if (w is '.'):
        print()
for w in reader.words(fileN):
    print(w + ' ', end='')
    if (w is '.'):
        print()
```

These for loops read every file one by one and will print on the console line by line. The output of the complete recipe should look similar to this:

```
['neg', 'pos']
['neg/cv000_29416.txt', 'neg/cv001_19502.txt',
'neg/cv002_17424.txt', ...]
pos/cv182_7281.txt
neg/cv712_24217.txt
the saint was actually a little better than i expected it to be ,
in some ways .
in this theatrical remake of the television series the saint...
```

How it works...

The quintessential ingredient of this recipe is the `CategorizedPlaintextCorpusReader` class of NLTK. Since we already know that the corpus we have downloaded is categorized, we only need provide appropriate arguments when creating the `reader` object. The implementation of the `CategorizedPlaintextCorpusReader` class internally takes care of loading the samples in appropriate buckets ('pos' and 'neg' in this case).

Counting all the wh words in three different genres in the Brown corpus

The Brown corpus is part of the NLTK data package. It's one of the oldest text corpora assembled at Brown University. It contains a collection of 500 texts broadly categorized in to 15 different genres/categories such as news, humor, religion, and so on. This corpus is a good use case to showcase the categorized plaintext corpus, which already has topics/concepts assigned to each of the texts (sometimes overlapping); hence, any analysis you do on it can adhere to the attached topic.

Getting ready

The objective of this recipe is to get you to perform a simple counting task on any given corpus. We will be using `nltk` library's `FreqDist` object for this purpose here, but more elaboration on the power of `FreqDist` will follow in the next recipe. Here, we will just concentrate on the application problem.

How to do it...

1. Create a new file named `BrownWH.py` and add the following `import` statements to begin:

```
import nltk
from nltk.corpus import brown
```

We have imported the `nltk` library and the Brown corpus.

2. Next up, we will check all the genres in the corpus and will pick any three categories from them to proceed with our task:

```
print(brown.categories())
```

The `brown.categories()` function call will return the list of all genres in the Brown corpus. When you run this line, you will see the following output:

```
['adventure', 'belles_lettres', 'editorial', 'fiction',
'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery',
'news', 'religion', 'reviews', 'romance', 'science_fiction']
```

3. Now let's pick three genres--fiction, humor and romance--from this list as well as the whwords that we want to count out from the text of these three genres:

```
genres = ['fiction', 'humor', 'romance']
whwords = ['what', 'which', 'how', 'why', 'when', 'where', 'who']
```

We have created a list containing the three picked genres and another list containing the seven whwords.



Your list can be longer or shorter depending on what do you consider as whwords.

4. Since we have the genres and the words we want to count in lists, we will be extensively using the for loop to iterate over them and optimize the number of lines of code. So first, we write a for iterator on the genres list:

```
for i in range(0, len(genres)): genre = genres[i]
print()
print("Analysing '"+ genre + "' wh words")
genre_text = brown.words(categories = genre)
```

These four lines of code will only start iterating on the list genres and load the entire text of each genre in the genre_text variable as a continuous list words.

5. Next up is a complex little statement where we will use the nltk library's FreqDist object. For now, let's understand the syntax and the broad-level output we will get from it:

```
fdist = nltk.FreqDist(genre_text)
```

FreqDist() accepts a list of words and returns an object that contains the map word and its respective frequency in the input word list. Here, the fdist object will contain the frequency of each of the unique words in the genre_text word list.

6. I'm sure you've already guessed what our next step is going to be. We will simply access the fdist object returned by FreqDist() and get the count of each of the wh words. Let's do it:

```
for wh in whwords:
    print(wh + ': ', fdist[wh], end=' ')
```

We are iterating over the `whwords` word list, accessing the `fdist` object with each of the `wh` words as index, getting back the frequency/count of all of them, and printing them out.

After running the complete program, you will get this output:

```
['adventure', 'belles_lettres', 'editorial', 'fiction',  
'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery',  
'news', 'religion', 'reviews', 'romance', 'science-fiction']  
  
Analysing 'fiction' wh words  
  
what: 128 which: 123 how: 54 why: 18 when: 133 where: 76 who: 103  
  
Analysing 'humor' wh words  
  
what: 36 which: 62 how: 18 why: 9 when: 52 where: 15 who: 48  
  
Analysing 'romance' wh words  
  
what: 121 which: 104 how: 60 why: 34 when: 126 where: 54 who: 89
```

How it works...

On analyzing the output, you can clearly see that we have the word count of all seven `wh` words for the three picked genres on our console. By counting the population of `wh` words, you can, to a degree, gauge whether the given text is high on relative clauses or question sentences. Similarly, you may have a populated ontology list of important words that you want to get a word count of to understand the relevance of the given text to your ontology. Counting word populations and analyzing distributions of counts is one of the oldest, simplest, and most popular tricks of the trade to start any kind of textual analysis.

Explore frequency distribution operations on one of the web and chat text corpus files

Web and chat text corpus is non-formal literature that, as the name implies, contains content from Firefox discussion forums, scripts of movies, wine reviews, personal advertisements, and overheard conversations. Our objective here in this recipe is to understand the use of frequency distribution and its features/functions.

Getting ready

In keeping with the objective of this recipe, we will run the frequency distribution on the personal advertising file inside `nltk.corpus.webtext`. Following that, we will explore the various functionalities of the `nltk.FreqDist` object such as the count of distinct words, 10 most common words, maximum-frequency words, frequency distribution plot, and tabulation.

How to do it...

1. Create a new file named `webtext.py` and add the following three lines to it:

```
import nltk
from nltk.corpus import webtext
print(webtext.fileids())
```

We just imported the required libraries and the `webtext` corpus; along with that, we also printed the constituent file's names. Run the program and you shall see the following output:

```
['firefox.txt', 'grail.txt', 'overheard.txt', 'pirates.txt',
'singles.txt', 'wine.txt']
```

2. Now we will select the file that contains personal advertisement data and run frequency distribution on it. Add the following three lines for it:

```
fileid = 'singles.txt'
wbt_words = webtext.words(fileid)
fdist = nltk.FreqDist(wbt_words)
```

`singles.txt` contains our target data; so, we loaded the words from that file in `wbt_words` and ran frequency distribution on it to get the `FreqDist` object `fdist`.

3. Add the following lines, which will show the most commonly appearing word (with the `fdist.max()` function) and the count of that word (with the `fdist[fdist.max()]` operation):

```
print('Count of the maximum appearing token "', fdist.max(), '" : ',
fdist[fdist.max()])
```


4. The following line will show us the count of distinct words in the bag of our frequency distribution using the `fdist.N()` function. Add the line in your code:

```
print('Total Number of distinct tokens in the bag : ', fdist.N())
```

5. Now let's find out the 10 most common words in the selected corpus bag. The function `fdist.most_common()` will do this for us. Add the following two lines in the code:

```
print('Following are the most common 10 words in the bag')
print(fdist.most_common(10))
```

6. Let us tabulate the entire frequency distribution using the `fdist.tabulate()` function. Add these lines in the code:

```
print('Frequency Distribution on Personal Advertisements')
print(fdist.tabulate())
```

7. Now we will plot the graph of the frequency distribution with cumulative frequencies using the `fdist.plot()` function:

```
fdist.plot(cumulative=True)
```

Let's run the program and see the output; we will discuss the same in the following section:

```
['firefox.txt', 'grail.txt', 'overheard.txt', 'pirates.txt',
'singles.txt', 'wine.txt']

Count of the maximum appearing token " , " : 539

Total Number of distinct tokens in the bag : 4867

Following are the most common 10 words in the bag

[(',', 539), ('.', 353), ('/', 110), ('for', 99), ('and', 74),
('to',

4), ('lady', 68), ('-', 66), ('seeks', 60), ('a', 52)]

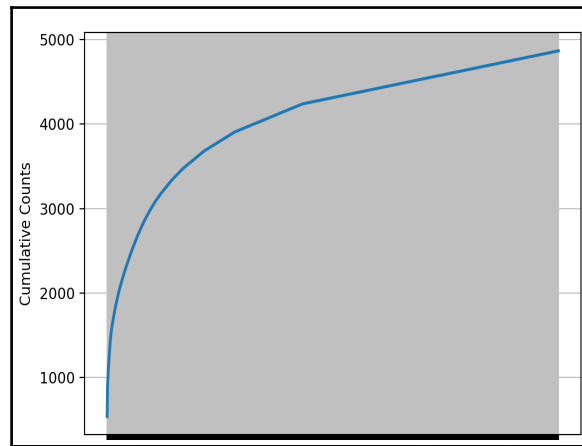
Frequency Distribution on Personal Advertisements

, . / for and to lady .....

539 353 110 99 74 74 .....

None
```

You will also see the following graph pop up:



How it works...

Upon analyzing the output, we realize that all of it is very intuitive. But what is peculiar is that most of it is not making sense. The token with maximum frequency count is , . And when you look at the 10 most common tokens, again you can't make out much about the target dataset. The reason is that there is no preprocessing done on the corpus. In the third chapter, we will learn one of the most fundamental preprocessing steps called stop words treatment and will also see the difference it makes.

Take an ambiguous word and explore all its senses using WordNet

From this recipe onwards, we will turn our attention to WordNet. As you can read in the title, we are going to explore what word sense is. To give an overview, English is a very ambiguous language. Almost every other word has a different meaning in different contexts. For example, let's take the simplest of words, *bat* which you will learn as part of the first 10 English words in a language course almost anywhere on the planet. The first meaning is a club used for hitting the ball in various sports such as cricket, baseball, tennis, squash, and so on.

Now a *bat* can also mean a nocturnal mammal that flies at nights. The *Bat* is also Batman's preferred and most advanced transportation vehicle according to DC comics. These are all noun variants; let's consider verb possibilities. *Bat* can also mean a slight wink (bat an eyelid). Consequently, it can also mean beating someone to pulp in a fight or a competition. We believe that's enough of an introduction; with this, let's move on to the actual recipe.

Getting ready

Keeping the objective of the recipe in mind we have to choose a word for which we would be exploring its various senses as understood by WordNet. And yes, NLTK comes equipped with WordNet; you need not worry about installing any further libraries. So let's choose another simple word, *CHAIR*, as our sample for the purpose of this recipe.

How to do it...

1. Create a new file named `ambiguity.py` and add the following lines of code to start with:

```
from nltk.corpus import wordnet as wn
chair = 'chair'
```

Here we imported the required NLTK corpus reader `wordnet` as the `wn` object. We can import it just like any another corpus readers we have used so far. In preparation for the next steps, we have created our string variable containing the word `chair`.

2. Now is the most important step. Let's add two lines and I will elaborate what we are doing:

```
chair_synsets = wn.synsets(chair)
print('Synsets/Senses of Chair :', chair_synsets, '\n\n')
```

The first line, though it looks simple, is actually the API interface that is accessing the internal WordNet database and fetching all the senses associated with the word `chair`. WordNet calls each of these senses `synsets`. The next line simply asks the interpreter to print what it has fetched. Run this much and you should get an output like:

```
Synsets/Senses of Chair : [Synset('chair.n.01'),
Synset('professorship.n.01'), Synset('president.n.04'),
Synset('electric_chair.n.01'), Synset('chair.n.05'),
Synset('chair.v.01'), Synset('moderate.v.01')]
```

As you can see, the list contains seven `Synsets`, which means seven different senses of the word `Chair` exist in the WordNet database.

3. We will add the following `for` loop, which will iterate over the list of `synsets` we have obtained and perform certain operations:

```
for synset in chair_synsets:
    print(synset, ': ')
    print('Definition: ', synset.definition())
    print('Lemmas/Synonymous words: ', synset.lemma_names())
    print('Example: ', synset.examples(), '\n')
```

We are iterating over the list of `synsets` and printing the definition of each sense, associated lemmas/synonymous words, and example usage of each of the senses in a sentence. One typical iteration will print something similar to this:

```
Synset('chair.v.01') :

Definition: act or preside as chair, as of an academic department
in a university

Lemmas/Synonymous words: ['chair', 'chairman']

Example: ['She chaired the department for many years']
```

The first line is the name of `Synset`, the second line is the definition of this sense/`Synset`, the third line contains `Lemmas` associated with this `Synset`, and the fourth line is an example sentence.

We will obtain this output:

```
Synsets/Senses of Chair : [Synset('chair.n.01'),  
Synset('professorship.n.01'), Synset('president.n.04'),  
Synset('electric_chair.n.01'), Synset('chair.n.05'),  
Synset('chair.v.01'), Synset('moderate.v.01')]
```

```
Synset('chair.n.01') :
```

```
Definition: a seat for one person, with a support for the back
```

```
Lemmas/Synonymous words: ['chair']
```

```
Example: ['he put his coat over the back of the chair and sat  
down']
```

```
Synset('professorship.n.01') :
```

```
Definition: the position of professor
```

```
Lemmas/Synonymous words: ['professorship', 'chair']
```

```
Example: ['he was awarded an endowed chair in economics']
```

```
Synset('president.n.04') :
```

```
Definition: the officer who presides at the meetings of an  
organization
```

```
Lemmas/Synonymous words: ['president', 'chairman', 'chairwoman',  
chair', 'chairperson']
```

```
Example: ['address your remarks to the chairperson']
```

```
Synset('electric_chair.n.01') :
```

```
Definition: an instrument of execution by electrocution; resembles  
an ordinary seat for one person
```

```
Lemmas/Synonymous words: ['electric_chair', 'chair', 'death_chair',  
'hot_seat']
```

```
Example: ['the murderer was sentenced to die in the chair']
```

```
Synset('chair.n.05') :
```

```
Definition: a particular seat in an orchestra
```

```
Lemmas/Synonymous words: ['chair']

Example: ['he is second chair violin']

Synset('chair.v.01') :

Definition: act or preside as chair, as of an academic department
in a university

Lemmas/Synonymous words: ['chair', 'chairman']

Example: ['She chaired the department for many years']

Synset('moderate.v.01') :

Definition: preside over

Lemmas/Synonymous words: ['moderate', 'chair', 'lead']

Example: ['John moderated the discussion']
```

How it works...

As you can see, definitions, Lemmas, and example sentences of all seven senses of the word `chair` are seen in the output. Straightforward API interfaces are available for each of the operations as elaborated in the preceding code sample. Now, let's talk a little bit about how WordNet arrives at such conclusions. WordNet is a database of words that stores all information about them in a hierarchical manner. If we take a look at the current example Write about `synsets` and hierarchical nature of WordNet storage. The following diagram will explain it in more detail.

Pick two distinct synsets and explore the concepts of hyponyms and hypernyms using WordNet

A hyponym is a word of a more specific meaning than a more generic word such as *bat*, which we explored in the introduction section of our previous recipe. What we mean by *more specific* is, for example, cricket bat, baseball bat, carnivorous bat, squash racket, and so on. These are more specific in terms of communicating what exactly we are trying to mean.

As opposed to a hyponym, a hypernym is a more general form or word of the same concept. For our example, *bat* is a more generic word and it could mean club, stick, artifact, mammal, animal, or organism. We can go as generic as the physical entity, living thing, or object and still be considered as a hypernym of the word *bat*.

Getting ready

For the purpose of exploring the concepts of hyponym and hypernym, we have decided to select the synsets `bed.n.01` (first word sense of bed) and `woman.n.01` (second word sense of woman). Now we will explain the usage and meaning of the hypernym and hyponym APIs in the actual recipe section.

How to do it...

1. Create a new file named `HypoNHypernyms.py` and add following three lines:

```
from nltk.corpus import wordnet as wn
woman = wn.synset('woman.n.02')
bed = wn.synset('bed.n.01')
```

We've imported the libraries and initialized the two synsets that we will use in later processing.

2. Add the following two lines:

```
print(woman.hypernyms())
woman_paths = woman.hypernym_paths()
```

It's a simple call to the `hypernyms()` API function on the `woman` `Synset`; it will return the set of synsets that are direct parents of the same. However, the `hypernym_paths()` function is a little tricky. It will return a list of sets. Each set contains the path from the root node to the `woman` `Synset`. When you run these two statements, you will see the two direct parents of the `Synset` `woman` as follows in the console:

```
[Synset('adult.n.01'), Synset('female.n.02')]
```

Woman belongs to the `adult` and `female` categories in the hierarchical structure of the WordNet database.

3. Now we will try to print the paths from root node to the `woman.n.01` node. To do so, add the following lines of code and nested `for` loop:

```
for idx, path in enumerate(woman_paths):
    print('\n\nHypernym Path :', idx + 1)
    for synset in path:
        print(synset.name(), ' ', end='')
```

As explained, the returned object is a list of sets ordered in such a way that it follows the path from the root to the `woman.n.01` node exactly as stored in the WordNet hierarchy. When you run, here's an example `Path`:

```
Hypernym Path : 1
```

```
entity.n.01 , physical_entity.n.01 , causal_agent.n.01 ,
person.n.01 , adult.n.01 , woman.n.01
```

4. Now let's work with `hyponyms`. Add the following two lines, which will fetch the `hyponyms` for the `synset` `bed.n.01` and print them to the console:

```
types_of_beds = bed.hyponyms()
print('\n\nTypes of beds(Hyponyms): ', types_of_beds)
```


As explained, run them and you will see the following 20 synsets as output:

```
Types of beds(Hyponyms): [Synset('berth.n.03'), Synset('built-
in_bed.n.01'), Synset('bunk.n.03'), Synset('bunk_bed.n.01'),
Synset('cot.n.03'), Synset('couch.n.03'), Synset('deathbed.n.02'),
Synset('double_bed.n.01'), Synset('four-poster.n.01'),
Synset('hammock.n.02'), Synset('marriage_bed.n.01'),
Synset('murphy_bed.n.01'), Synset('plank-bed.n.01'),
Synset('platform_bed.n.01'), Synset('sickbed.n.01'),
Synset('single_bed.n.01'), Synset('sleigh_bed.n.01'),
Synset('trundle_bed.n.01'), Synset('twin_bed.n.01'),
Synset('water_bed.n.01')]
```

These are Hyponyms or more specific terms for the word sense `bed.n.01` within WordNet.

- Now let's print the actual words or lemmas that will make more sense to humans. Add the following line of code:

```
print(sorted(set(lemma.name() for synset in types_of_beds for lemma
in synset.lemmas())))
```

This line of code is pretty similar to what we did in the hypernym example nested `for` loop written in four lines, which is clubbed in a single line here (in other words, we're just showing off our skills with Python here). It will print the 26 lemmas that are very meaningful and specific words. Now let's look at the final output:

```
Output: [Synset('adult.n.01'), Synset('female.n.02')]
Hypernym Path : 1
entity.n.01 , physical_entity.n.01 , causal_agent.n.01 ,
person.n.01 , adult.n.01 , woman.n.01 ,
Hypernym Path : 2
entity.n.01 , physical_entity.n.01 , object.n.01 , whole.n.02 ,
living_thing.n.01 , organism.n.01 , person.n.01 , adult.n.01 ,
woman.n.01 ,
Hypernym Path : 3
entity.n.01 , physical_entity.n.01 , causal_agent.n.01 ,
person.n.01 , female.n.02 , woman.n.01 ,
Hypernym Path : 4
entity.n.01 , physical_entity.n.01 , object.n.01 , whole.n.02 ,
living_thing.n.01 , organism.n.01 , person.n.01 , female.n.02 ,
woman.n.01 ,

Types of beds(Hyponyms): [Synset('berth.n.03'), Synset('built-
in_bed.n.01'), Synset('bunk.n.03'), Synset('bunk_bed.n.01'),
Synset('cot.n.03'), Synset('couch.n.03'), Synset('deathbed.n.02'),
```

```

Synset('double_bed.n.01'), Synset('four-poster.n.01'),
Synset('hammock.n.02'), Synset('marriage_bed.n.01'),
Synset('murphy_bed.n.01'), Synset('plank-bed.n.01'),
Synset('platform_bed.n.01'), Synset('sickbed.n.01'),
Synset('single_bed.n.01'), Synset('sleigh_bed.n.01'),
Synset('trundle_bed.n.01'), Synset('twin_bed.n.01'),
Synset('water_bed.n.01')]

['Murphy_bed', 'berth', 'built-in_bed', 'built_in_bed', 'bunk',
'bunk_bed', 'camp_bed', 'cot', 'couch', 'deathbed', 'double_bed',
'four-poster', 'hammock', 'marriage_bed', 'plank-bed',
'platform_bed', 'sack', 'sickbed', 'single_bed', 'sleigh_bed',
'truckle', 'truckle_bed', 'trundle', 'trundle_bed', 'twin_bed',
'water_bed']

```

How it works...

As you can see, `woman.n.01` has two hypernyms, namely `adult` and `female`, but it follows four different routes in the hierarchy of WordNet database from the root node `entity` to `woman` as shown in the output.

Similarly, the Synset `bed.n.01` has 20 hyponyms; they are more specific and less ambiguous (for nothing is unambiguous in English). Generally the hyponyms correspond to leaf nodes or nodes very much closer to the leaves in the hierarchy as they are the least ambiguous ones.

Compute the average polysemy of nouns, verbs, adjectives, and adverbs according to WordNet

First, let's understand what polysemy is. Polysemy means many possible meanings of a word or a phrase. As we have already seen, English is an ambiguous language and more than one meaning usually exists for most of the words in the hierarchy. Now, turning back our attention to the problem statement, we must calculate the average polysemy based on specific linguistic properties of all words in WordNet. As we'll see, this recipe is different from previous recipes. It's not just an API concept discovery but we are going to discover a linguistic concept here (I'm all emotional to finally get a chance to do so in this chapter).

Getting ready

I have decided to write the program to compute the polysemy of any one of the POS types of words and will leave it to you guys to modify the program to do so for the other three. I mean we shouldn't just spoon-feed everything, right? Not to worry! I will provide enough hints in the recipe itself to make it easier for you (for those who think it's already not very intuitive). Let's get on with the actual recipe then; we will compute the average polysemy of nouns alone.

How to do it...

1. Create a new file named `polysemy.py` and add these two initialization lines:

```
from nltk.corpus import wordnet as wn
type = 'n'
```

We have initialized the POS type of words we are interested in and, of course, imported the required libraries. To be more descriptive, `n` corresponds to nouns.

2. This is the most important line of code of this recipe:

```
synsets = wn.all_synsets(type)
```

This API returns all `synsets` of type `n` that is a noun present in the WordNet database, full coverage. Similarly, if you change the POS type to a verb, adverb, or adjective, the API will return all words of the corresponding type (hint #1).

3. Now we will consolidate all `lemmas` in each of the `synset` into a single mega list that we can process further. Add the following code to do that:

```
lemmas = []
for synset in synsets:
    for lemma in synset.lemmas():
        lemmas.append(lemma.name())
```

This piece of code is pretty intuitive; we have a nested `for` loop that iterates over the list of `synsets` and the `lemmas` in each `synset` and adds them up in our mega list `lemmas`.

4. Although we have all `lemmas` in the mega list, there is a problem. There are some duplicates as it's a list. Let's remove the duplicates and take the count of distinct lemmas:

```
lemmas = set(lemmas)
```

Converting a list into a set will automatically deduplicate (yes, it's a valid English word, I invented it) the list.

5. Now, the second most important step in the recipe. We count the senses of each lemma in the WordNet database:

```
count = 0
for lemma in lemmas:
    count = count + len(wn.synsets(lemma, type))
```

Most of the code is intuitive; let's focus on the the API `wn.synsets(lemma, type)`. This API takes as input a word/lemma (as the first argument) and the POS type it belongs to and returns all the senses (`synsets`) belonging to the lemma word. Note that depending on what you provide as the POS type, it will return senses of the word of only the given POS type (hint #2).

6. We have all the counts we need to compute the average polysemy. Let's just do it and print it on the console:

```
print('Total distinct lemmas: ', len(lemmas))
print('Total senses :', count)
print('Average Polysemy of ', type, ': ' , count/len(lemmas))
```

This prints the total distinct lemmas, the count of senses, and the average polysemy of POS type `n` or nouns:

```
Output: Total distinct lemmas: 119034
Total senses : 152763
Average Polysemy of n : 1.2833560159282222
```

How it works...

There is nothing much to say in this section, so I will instead give you some more information on how to go about computing the polysemy of the rest of the types. As you saw, *Noun* -> `'n'`. Similarly, *Verbs* -> `'v'`, *Adverbs* -> `'r'`, and *Adjective* -> `'a'` (hint # 3).

Now, I hope I have given you enough hints to get on with writing an NLP program of your own and not be dependent on the feed of the recipes.

2

Raw Text, Sourcing, and Normalization

In this chapter, we will be covering the following topics:

- The importance of string operations
- Getting deeper with string operations
- Reading a PDF file in Python
- Reading Word documents in Python
- Taking PDF, DOCX, and plain text files and creating a user-defined corpus from them
- Reading contents from an RSS feed
- HTML parsing using BeautifulSoup

Introduction

In the previous chapter, we looked at NLTK inbuilt corpora. The corpora are very well organized and normalized for usage, but that will not always be the case when you work on your industry problems. Let alone normalization and organization, we may not even get the data we need in a uniform format. The goal of this chapter is to introduce some Python libraries that will help you extract data from binary formats: PDF and Word DOCX files. We will also look at libraries that can fetch data from web feeds such as RSS and a library that will help you parse HTML and extract the raw text out of the documents. We will also learn to extract raw text from heterogeneous sources, normalize it, and create a user-defined corpus from it.

In this chapter, you will learn seven different recipes. As the name of the chapter suggests, we will be learning to source data from PDF files, Word documents, and the Web. PDFs and Word documents are binary, and over the Web, you will get data in the form of HTML. For this reason, we will also perform normalization and raw text conversion tasks on this data.

The importance of string operations

As an NLP expert, you are going to work on a lot of textual content. And when you are working with text, you must know string operations. We are going to start with a couple of short and crisp recipes that will help you understand the `str` class and operations with it in Python.

Getting ready...

For this recipe, you will just need the Python interpreter and a text editor, nothing more. We will see `join`, `split`, `addition`, and `multiplication` operators and indices.

How to do it...

1. Create a new Python file named `StringOps1.py`.
2. Define two objects:

```
namesList = ['Tuffy', 'Ali', 'Nysha', 'Tim' ]  
sentence = 'My dog sleeps on sofa'
```

The first object, `nameList`, is a list of `str` objects containing some names as implied, and the second object, `sentence`, is a sentence that is an `str` object.

3. First, we will see the `join` functionality and what it does:

```
names = ';'.join(namesList)  
print(type(names), ': ', names)
```

The `join()` function can be called on any `string` object. It accepts a list of `str` objects as argument and concatenates all the `str` objects into a single `str` object, with the calling `string` object's contents as the joining delimiter. It returns that object. Run these two lines and your output should look like:

```
<class 'str'> : Tuffy;Ali;Nysha;Tim
```

4. Next, we will check out the `split` method:

```
wordList = sentence.split(' ')
print((type(wordList)), ':', wordList)
```

The `split` function called on a string will split its contents into multiple `str` objects, create a list of the same, and return that list. The function accepts a single `str` argument, which is used as the splitting criterion. Run the code and you will see the following output:

```
<class 'list'> : ['My', 'dog', 'sleeps', 'on', 'sofa']
```

5. The arithmetic operators `+` and `*` can also be used with strings. Add the following lines and see the output:

```
additionExample = 'ganehsa' + 'ganesha' + 'ganesha'
multiplicationExample = 'ganesha' * 2
print('Text Additions :', additionExample)
print('Text Multiplication :', multiplicationExample)
```

This time we will first see the output and then discuss how it works:

```
Text Additions: ganehsaganesganesha
Text Multiplication: ganesganesha
```

The `+` operator is known as concatenation. It produces a new string, concatenating the strings into a single `str` object. Using the `*` operator, we can multiply the strings too, as shown previously in the output. Also, please note that these operations don't add anything extra, such as insert a space between the strings.

6. Let's look at the indices of the characters in the strings. Add the following lines of code:

```
str = 'Python NLTK'
print(str[1])
print(str[-3])
```

First, we declare a new `string` object. Then we access the second character (`y`) in the string, which just shows that it is straightforward. Now comes the tricky part; Python allows you to use negative indexes when accessing any list object; `-1` means the last member, `-2` is the second last, and so on. For example, in the preceding `str` object, index 7 and `-4` are the same character, `N`:

```
Output: <class 'str'> : Tuffy;Ali;Nysha;Tim
<class 'list'> : ['My', 'dog', 'sleeps', 'on', 'sofa']
Text Additions : ganehsaganeshaganesha
Text Multiplication : ganeshaganesha
Y
L
```

How it works...

We created a list of strings from a string and a string from a list of strings using the `split()` and `join()` functions, respectively. Then we saw the use of some arithmetic operators with strings. Please note that we can't use the `"-"` (negation) and the `"/"` (division) operators with strings. In the end, we saw how to access individual characters in any string, in which peculiarly, we can use negative index numbers while accessing strings.

This recipe is pretty simple and straightforward, in that the objective was to introduce some common and uncommon string operations that Python allows. Up next, we will continue where we left off and do some more string operations.

Getting deeper with string operations

Moving ahead from the previous recipe, we will see substrings, string replacements, and how to access all the characters of a string.

Let's get started.

How to do it...

1. Create a new Python file named `StringOps2.py` and define the following string object `str`:

```
str = 'NLTK Dolly Python'
```


2. Let's access the substring that ends at the fourth character from the `str` object.

```
print('Substring ends at:',str[:4])
```

As we know the index starts at zero, this will return the substring containing characters from zero to three. When you run, the output will be:

```
Substring ends at: NLTK
```

3. Now we will access the substring that starts at a certain point until the end in object `str`:

```
print('Substring starts from:',str[11:] )
```

This tells the interpreter to return a substring of object `str` from index 11 to the end. When you run this, the following output will be visible:

```
Substring starts from: Python
```

4. Let's access the Dolly substring from the `str` object. Add the following line:

```
print('Substring :',str[5:10])
```

The preceding syntax returns characters from index 5 to 10, excluding the 10th character. The output is:

```
Substring : Dolly
```

5. Now, it's time for a fancy trick. We have already seen how negative indices work for string operations. Let's try the following and see how it works:

```
print('Substring fancy:', str[-12:-7])  
Run and check the output, it will be -  
Substring fancy: Dolly
```

Exactly similar to the previous step! Go ahead and do the back calculations: -1 as the last character, -2 as the last but one, and so and so forth. Thus, you will get the index values.

6. Let's check the `in` operator with `if`:

```
if 'NLTK' in str:  
    print('found NLTK')
```

Run the preceding code and check the output; it will be:

```
found NLTK
```

As elaborate as it looks, the `in` operator simply checks whether the left-hand side string is a substring of the right-hand side string.

7. We will use the simple `replace` function on an `str` object:

```
replaced = str.replace('Dolly', 'Dorothy')
print('Replaced String:', replaced)
```

The `replace` function simply takes two arguments. The first is the substring that needs to be replaced and the second is the new substring that will come in place of it. It returns a new `string` object and doesn't modify the object it was called upon. Run and see the following output:

```
Replaced String: NLTK Dorothy Python
```

8. Last but not least, we will iterate over the `replaced` object and access every character:

```
print('Accessing each character:')
for s in replaced:
    print(s)
```

This will print each character from the `replaced` object on a new line. Let's see the final output:

```
Output: Substring ends at: NLTK
Substring starts from: Python
Substring : Dolly
Substring fancy: Dolly
found NLTK
Replaced String: NLTK Dorothy Python
Accessing each character:
N
L
T
K
D
o
r
o
t
h
y
P
y
t
h
```

o
n

How it works...

A `string` object is nothing but a list of characters. As we saw in the first step we can access every character from the string using the `for` syntax for accessing a list. The character `:` inside square brackets for any list denotes that we want a piece of the list; `:` followed by a number means we want the sublist starting at zero and ending at the index minus 1. Similarly, a number followed by a `:` means we want a sublist from the given number to the end.

This ends our brief journey of exploring string operations with Python. After this, we will move on to files, online resources, HTML, and more.

Reading a PDF file in Python

We start off with a small recipe for accessing PDF files from Python. For this, you need to install the `PyPDF2` library.

Getting ready

We assume you have `pip` installed. Then, to install the `PyPDF2` library with `pip` on Python 2 and 3, you only need to run the following command from the command line:

```
pip install pypdf2
```

If you successfully install the library, we are ready to go ahead. Along with that, I also that request you to download some test documents that we will be using during this chapter from this link: <https://www.dropbox.com/sh/bk18dizhsu1p534/AABEuJw4TArUbzJf4Aa8gp5Wa?dl=0>.

How to do it...

1. Create a file named `pdf.py` and add the following import line to it:

```
from PyPDF2 import PdfFileReader
```

It imports the `PdfFileReader` class from the lib `PyPDF2`.

2. Add this Python function in the file that is supposed to read the file and return the full text from the PDF file:

```
def getTextPDF(pdfFileName, password = '')
```

This function accepts two arguments, the path to the PDF file you want to read and the password (if any) for the PDF file. As you can see, the `password` parameter is optional.

3. Now let's define the function. Add the following lines under the function:

```
pdf_file = open(pdfFileName, 'rb')
read_pdf = PdfFileReader(pdf_file)
```

The first line opens the file in read and backwards seek mode. The first line is essentially the Python open file command/function that will only open a file that is non-text in binary mode. The second line will pass this opened file to the `PdfFileReader` class, which will consume the PDF document.

4. The next step is to decrypt password-protected files, if any:

```
if password != '':
    read_pdf.decrypt(password)
```

If a password is provided with the function call, then we will try to decrypt the file using the same.

5. Now we will read the text from the file:

```
text = []
for i in range(0, read_pdf.getNumPages()-1):
    text.append(read_pdf.getPage(i).extractText())
```

We create a list of strings and append text from each page to that list of strings.

6. Return the final output:

```
return '\n'.join(text)
```

We return the single `string` object by joining the contents of all the string objects inside the list with a new line.

7. Create another file named `TestPDFs.py` in the same folder as `pdf.py`, and add the following import statement:

```
import pdf
```

8. Now we'll just print out the text from a couple of documents, one password protected and one plain:

```
pdfFile = 'sample-one-line.pdf'
pdfFileEncrypted = 'sample-one-line.encrypted.pdf'
print('PDF 1: \n',pdf.getTextPDF(pdfFile))
print('PDF 2: \n',pdf.getTextPDF(pdfFileEncrypted,'tuffy'))
```

Output: The first six steps of the recipe only create a Python function and no output will be generated on the console. The seventh and eighth steps will output the following:

```
This is a sample PDF document I am using to demonstrate in the
tutorial.
```

```
This is a sample PDF document
```

```
password protected.
```

How it works...

`PyPDF2` is a pure Python library that we use to extract content from PDFs. The library has many more functionalities to crop pages, superimpose images for digital signatures, create new PDF files, and much more. However, your purpose as an NLP engineer or in any text analytics task would only be to read the contents. In step 2, it's important to open the file in backwards seek mode since the `PyPDF2` module tries to read files from the end when loading the file contents. Also, if any PDF file is password protected and you do not decrypt it before accessing its contents, the Python interpreter will throw a `PdfReadError`.

Reading Word documents in Python

In this recipe, we will see how to load and read Word/DOCX documents. The libraries available for reading DOCX word documents are more comprehensive, in that we can also see paragraph boundaries, text styles, and do what are called runs. We will see all of this as it can prove vital in your text analytics tasks.



If you do not have access to Microsoft Word, you can always use open source versions of Liber Office and Open Office to create and edit .docx files.

Getting ready...

Assuming you already have `pip` installed on your machine, we will use `pip` to install a module named `python-docx`. Do not confuse this with another library named `docx`, which is a different module altogether. We will be importing the `docx` object from the `python-docx` library. The following command, when fired on your command line, will install the library:

```
pip install python-docx
```

After having successfully installed the library, we are ready to go ahead. We will be using a test document in this recipe, and if you have already downloaded all the documents from the link shared in the first recipe in this chapter, you should have the relevant document. If not, then please download the `sample-one-line.docx` document from <https://www.dropbox.com/sh/bk18dizhsulp534/AABEuJw4TArUbzJf4Aa8gp5Wa?dl=0>.

Now we are good to go.

How to do it...

1. Create a new Python file named `word.py` and add the following `import` line:

```
import docx
```

Simply import the `docx` object of the `python-docx` module.

2. Define the function `getTextWord`:

```
def getTextWord(wordFileName):
```

The function accepts one `string` parameter, `wordFileName`, which should contain the absolute path to the Word file you are interested in reading.

3. Initialize the `doc` object:

```
doc = docx.Document(wordFileName)
```

The `doc` object is now loaded with the word file you want to read.

4. We will read the text from the document loaded inside the `doc` object. Add the following lines for that:

```
fullText = []
for para in doc.paragraphs:
    fullText.append(para.text)
```

First, we initialized a string array, `fullText`. The `for` loop reads the text from the document paragraph by paragraph and goes on appending to the list `fullText`.

5. Now we will join all the fragments/paras in a single string object and return it as the final output of the function:

```
return '\n'.join(fullText)
```

We joined all the constituents of the `fullText` array with the delimited `\n` and returned the resultant object. Save the file and exit.

6. Create another file, name it `TestDocX.py`, and add the following import statements:

```
import docx
import word
```

Simply import the `docx` library and the `word.py` that we wrote in the first five steps.

7. Now we will read a DOCX document and print the full contents using the API we wrote on `word.py`. Write down the following two lines:

```
docName = 'sample-one-line.docx'
print('Document in full :\n',word.getTextWord(docName))
```

Initialize the document path in the first line, and then, using the API print out the full document. When you run this part, you should get an output that looks something similar to:

Document in full :

This is a sample PDF document with some text in bold, some in italic, and some underlined. We are also embedding a title shown as follows:

This is my TITLE.
This is my third paragraph.

8. As already discussed, Word/DOCX documents are a much richer source of information and the libraries will give us much more than text. Now let us look at the paragraph information. Add the following four lines of code:

```
doc = docx.Document(docName)
print('Number of paragraphs :',len(doc.paragraphs))
print('Paragraph 2:',doc.paragraphs[1].text)
print('Paragraph 2 style:',doc.paragraphs[1].style)
```

The second line in the previous snippet gives us the number of paragraphs in the given document. The third line returns only the second paragraph from the document and the fourth line will analyze the style of the second paragraph, which is `Title` in this case. When you run, the output for these four lines will be:

Number of paragraphs : 3
Paragraph 2: This is my TITLE.
Paragraph 2 style: _ParagraphStyle('Title') id: 4374023248

It is quite self-explanatory.

9. Next, we will see what a run is. Add the following lines:

```
print('Paragraph 1:',doc.paragraphs[0].text)
print('Number of runs in paragraph 1:',len(doc.paragraphs[0].runs))
for idx, run in enumerate(doc.paragraphs[0].runs):
    print('Run %s : %s' %(idx,run.text))
```


Here, we are first returning the first paragraph; next we are returning the number of runs in the paragraph. Later we are printing out every run.

10. And now to identify the styling of each run, write the following lines of code:

```
print('is Run 0 underlined:', doc.paragraphs[0].runs[5].underline)
print('is Run 2 bold:', doc.paragraphs[0].runs[1].bold)
print('is Run 7 italic:', doc.paragraphs[0].runs[3].italic)
```

Each line in the previous snippet is checking for underline, bold, and italic styling respectively. In the following section, we will see the final output:

```
Output: Document in full :
This is a sample PDF document with some text in BOLD, some in
ITALIC and some underlined. We are also embedding a Title down
below.
This is my TITLE.
This is my third paragraph.
Number of paragraphs : 3
Paragraph 2: This is my TITLE.
Paragraph 2 style: _ParagraphStyle('Title') id: 4374023248
Paragraph 1: This is a sample PDF document with some text in BOLD,
some in ITALIC and some underlined. We're also embedding a Title
down below.
Number of runs in paragraph 1: 8
Run 0 : This is a sample PDF document with
Run 1 : some text in BOLD
Run 2 : ,
Run 3 : some in ITALIC
Run 4 : and
Run 5 : some underlined.
Run 6 : We are also embedding a Title down below
Run 7 : .
is Run 0 underlined: True
is Run 2 bold: True
is Run 7 italic: True
```

How it works...

First, we wrote a function in the `word.py` file that will read any given DOCX file and return to us the full contents in a `string` object. The preceding output text you see is fairly self-explanatory though some things I would like to elaborate are `Paragraph` and `Run` lines. The structure of a `.docx` document is represented by three data types in the `python-docx` library. At the highest level is the `Document` object. Inside each document, we have multiple paragraphs.

Every time we see a new line or a carriage return, it signifies the start of a new paragraph. Every paragraph contains multiple `Runs`, which denotes a change in word styling. By styling, we mean the possibilities of different fonts, sizes, colors, and other styling elements such as bold, italic, underline, and so on. Each time any of these elements vary, a new run is started.

Taking PDF, DOCX, and plain text files and creating a user-defined corpus from them

For this recipe, we are not going to use anything new in terms of libraries or concepts. We are reinvoking the concept of corpus from the first chapter. Just that we are now going to create our own corpus here instead of using what we got from the Internet.

Getting ready

In terms of getting ready, we are going to use a few files from the Dropbox folder introduced in the first recipe of this chapter. If you've downloaded all the files from the folder, you should be good. If not, please download the following files from <https://www.dropbox.com/sh/bk18dizhsu1p534/AABEuJw4TArUbzJf4Aa8gp5Wa?dl=0>:

- `sample_feed.txt`
- `sample-pdf.pdf`
- `sample-one-line.docx`

If you haven't followed the order of this chapter, you will have to go back and look at the first two recipes in this chapter. We are going to reuse two modules we wrote in the previous two recipes, `word.py` and `pdf.py`. This recipe is more about an application of what we did in the first two recipes and the corpus from the first chapter than introducing a new concept. Let's get on with the actual code.

How to do it...

1. Create a new Python file named `createCorpus.py` and add the following import lines to start off:

```
import os
import word, pdf
from nltk.corpus.reader.plaintext import PlaintextCorpusReader
```

We have imported the `os` library for use with file operations, the `word` and `pdf` modules we wrote in the first two recipes of this chapter, and the `PlaintextCorpusReader`, which is our final objective of this recipe.

2. Now let's write a little function that will take as input the path of a plain text file and return the full text as a `string` object. Add the following lines:

```
def getText(txtFileName):
    file = open(txtFileName, 'r')
    return file.read()
```

The first line defines the function and input parameter. The second line opens the given file in reading mode (the second parameter of the `open` function `r` denotes read mode). The third line reads the content of the file and returns it into a `string` object, all at once in a single statement.

3. We will create the new `corpus` folder now on the disk/filesystem. Add the following three lines:

```
newCorpusDir = 'mycorpus/'
if not os.path.isdir(newCorpusDir):
    os.mkdir(newCorpusDir)
```

The first line is a simple `string` object with the name of the new folder. The second line checks whether a directory/folder of the same name already exists on the disk. The third line instructs the `os.mkdir()` function to create the directory on the disk with the specified name. As the outcome, a new directory with the name `mycorpus` would be created in the working directory where your Python file is placed.

4. Now we will read the three files one by one. Starting with the plain text file, add the following line:

```
txt1 = getText('sample_feed.txt')
```

Calling the `getText()` function written earlier, it will read the `sample_feed.txt` file and return the output in the `txt1` string object.

5. Now we will read the PDF file. Add the following line:

```
txt2 = pdf.getTextPDF('sample-pdf.pdf')
```

Using the `pdf.py` module's `getTextPDF()` function, we are retrieving the contents of the `sample-pdf.pdf` file into the `txt2` string object.

6. Finally, we will read the DOCX file by adding the following line:

```
txt3 = word.getTextWord('sample-one-line.docx')
```

Using the `word.py` module's `getTextWord()` function, we are retrieving the contents of the `sample-one-line.docx` file into the `txt3` string object.

7. The next step is to write the contents of these three string objects on the disk, in files. Write the following lines of code for that:

```
files = [txt1,txt2,txt3]
for idx, f in enumerate(files):
    with open(newCorpusDir+str(idx)+'.txt', 'w') as fout:
        fout.write(f)
```

- **First line:** Creates an array from the string objects so as to use it in the upcoming for loop
- **Second line:** A for loop with index on the files array
- **Third line:** This opens a new file in write mode (the `w` option in the open function call)
- **Fourth line:** Writes the contents of the string object in the file

8. Now we will create a `PlainTextCorpus` object from the `mycorpus` directory, where we have stored our files:

```
newCorpus = PlaintextCorpusReader(newCorpusDir, '.*')
```

A simple one-line instruction but internally it does a lot of text processing, identifying paragraphs, sentences, words, and much more. The two parameters are the path to the corpus directory and the pattern of the filenames to consider (here we have asked the corpus reader to consider all files in the directory). We have created a user-defined corpus. As simple as that!

9. Let us see whether the our `PlainTextCorpusReader` is loaded correctly. Add the following lines of code to test it:

```
print(newCorpus.words())
print(newCorpus.sents(newCorpus.fileids()[1]))
print(newCorpus.paras(newCorpus.fileids()[0]))
```

The first line will print the array containing all the words in the corpus (curtailed). The second line will print the sentences in file `1.txt`. The third line will print the paragraphs in file `0.txt`:

```
Output: ['Five', 'months', '.', 'That', '"', 's', 'how', ...]
[['A', 'generic', 'NLP'], [('(', 'Natural', 'Language',
'Processing', ')', 'toolset'], ...]
[[['Five', 'months', '.']], [['That', '"', 's', 'how', 'long',
'it', '"', 's', 'been', 'since', 'Mass', 'Effect', ':',
'Andromeda', 'launched', ',', 'and', 'that', '"', 's', 'how',
'long', 'it', 'took', 'BioWare', 'Montreal', 'to', 'admit', 'that',
'nothing', 'more', 'can', 'be', 'done', 'with', 'the', 'ailing',
'game', '"', 's', 'story', 'mode', '.'], ['Technically', ',', 'it',
'wasn', '"', 't', 'even', 'a', 'full', 'five', 'months', ',', 'as',
'Andromeda', 'launched', 'on', 'March', '21', '.']], ...]
```

How it works...

The output is fairly straightforward and as explained in the last step of the recipe. What is peculiar is the characteristics of each of objects on show. The first line is the list of all words in the new corpus; it doesn't have anything to do with higher level structures like sentences/paragraphs/files and so on. The second line is the list of all sentences in the file `1.txt`, of which each sentence is a list of words inside each of the sentences. The third line is a list of paragraphs, of which each paragraph object is in turn a list of sentences, of which each sentence is in turn a list of words in that sentence, all from the file `0.txt`. As you can see, a lot of structure is maintained in paragraphs and sentences.

Read contents from an RSS feed

A **Rich Site Summary (RSS)** feed is a computer-readable format in which regularly changing content on the Internet is delivered. Most of the websites that provide information in this format give updates, for example, news articles, online publishing and so on. It gives the listeners access to the updated feed at regular intervals in a standardized format.

Getting ready

The objective of this recipe is to read such an RSS feed and access content of one of the posts from that feed. For this purpose, we will be using the RSS feed of Mashable. Mashable is a digital media website, in short a tech and social media blog listing. The URL of the website's RS feed is `http://feeds.mashable.com/Mashable`.

Also, we need the `feedparser` library to be able to read an RSS feed. To install this library on your computer, simply open the terminal and run the following command:

```
pip install feedparser
```

Armed with this module and the useful information, we can begin to write our first RSS feed reader in Python.

How to do it...

1. Create a new file named `rssReader.py` and add the following import:

```
import feedparser
```

2. Now we will load the Mashable feed into our memory. Add the following line:

```
myFeed = feedparser.parse("http://feeds.mashable.com/Mashable")
```

The `myFeed` object contains the first page of the RSS feed of Mashable. The feed will be downloaded and parsed to fill all the appropriate fields by the `feedparser`. Each post will be part of the entry list in to the `myFeed` object.

3. Let's check the title and count the number of posts in the current feed:

```
print('Feed Title :', myFeed['feed']['title'])  
print('Number of posts :', len(myFeed.entries))
```

In the first line, we are fetching the feed title from the `myFeed` object, and in the second line, we are counting the length of the `entries` object inside the `myFeed` object. The `entries` object is nothing but a list of all the posts from the parsed feed as mentioned previously. When you run, the output is something similar to:

```
Feed Title: Mashable  
Number of posts : 30
```

Title will always be Mashable, and at the time of writing this chapter, the Mashable folks were putting a maximum of 30 posts in the feed at a time.

4. Now we will fetch the very first `post` from the `entries` list and print its title on the console:

```
post = myFeed.entries[0]  
print('Post Title :',post.title)
```

In the first line, we are physically accessing the zeroth element in the `entries` list and loading it in the `post` object. The second line prints the title of that post. Upon running, you should get an output similar to the following:

```
Post Title: The moon literally blocked the sun on Twitter
```

I say something similar and not exactly the same as the feed keeps updating itself.

5. Now we will access the raw HTML content of the post and print it on the console:

```
content = post.content[0].value  
print('Raw content :\n',content)
```

First we access the `content` object from the `post` and the actual value of the same. And then we print it on the console:

```
Output: Feed Title: Mashable  
Number of posts : 30  
Post Title: The moon literally blocked the sun on Twitter  
Raw content :  
<div style="float: right;  
width: 50px;"><a  
href="http://twitter.com/share?via=Mashable&text=The+moon+literally  
+blocked+the+sun+on+Twitter&url=http%3A%2F%2Fmashable.com%2F2017%2F  
08%2F21%2Fmoon-blocks-sun-eclipse-2017-  
twitter%2F%3Futm_campaign%3DMash-Prod-RSS-Feedburner-All-
```

```
Partial%26utm_cid%3DMash-Prod-RSS-Feedburner-All-Partial"
style="margin: 10px;">
<p>The national space agency threw shade the best way it knows how:
by blocking the sun. Yep, you read that right. </p>
<div><div><blockquote>
<p>HA HA HA I've blocked the Sun! Make way for the Moon<a
href="https://twitter.com/hashtag/SolarEclipse2017?src=hash">#Solar
Eclipse2017</a> <a
href="https://t.co/nZCoqBlStE">pic.twitter.com/nZCoqBlStE</a></p>
<p>— NASA Moon (@NASAMoon) <a
href="https://twitter.com/NASAMoon/status/899681358737539073">August
t 21, 2017</a></p>
</blockquote></div></div>
```

How it works...

Most of the RSS feeds you will get on the Internet will follow a chronological order, with the latest post on top. Hence, the post we accessed in the recipe will be always be the most recent post the feed is offering. The feed itself is ever-changing. So every time you run the program, the format of the output will the remain same, but the content of the post on the console may differ depending upon how fast the feed updates. Also, here we are directly displaying the raw HTML on the console and not the clean content. Up next, we are going to look at parsing HTML and getting only the information we need from a page. Again, a further addendum to this recipe could be to read any feed of your choice, store all the posts from the feed on disk, and create a plain text corpus using it. Needless to say, you can take inspiration from the previous and the next recipes.

HTML parsing using BeautifulSoup

Most of the times when you have to deal with data on the Web, it will be in the form of HTML pages. For this purpose, we thought it is necessary to introduce you to HTML parsing in Python. There are many Python modules available to do this, but in this recipe, we will see how to parse HTML using the library `BeautifulSoup4`.

Getting ready

The package `BeautifulSoup4` will work for Python 2 and Python 3. We will have to download and install this package on our interpreter before we can start using it. In tune with what we have been doing throughout, we will use the `pip` install utility for it. Run the following command from the command line:

```
pip install beautifulsoup4
```

Along with this module, you will also need the `sample-html.html` file from the chapter's Dropbox location. In case you haven't downloaded the files already, here's the link again:

<https://www.dropbox.com/sh/bk18dizhsulp534/AABEuJw4TArUbzJf4Aa8gp5Wa?dl=0>

How to do it...

1. Assuming you have already installed the required package, start with the following import statement:

```
from bs4 import BeautifulSoup
```

We have imported the `BeautifulSoup` class from the module `bs4`, which we will be using to parse the HTML.

2. Let's load the HTML file into the `BeautifulSoup` object:

```
html_doc = open('sample-html.html', 'r').read()
soup = BeautifulSoup(html_doc, 'html.parser')
```

In the first line, we load the `sample-html.html` file's content into the `str` object `html_doc`. Next we create a `BeautifulSoup` object, passing to it the contents of our HTML file as the first argument and `html.parser` as the second argument. We instruct it to parse the document using the `html` parser. This will load the document into the `soup` object, parsed and ready to use.

3. The first, simplest, and most useful task on this `soup` object will be to strip all the HTML tags and get the text content. Add the following lines of code:

```
print('\n\nFull text HTML Stripped:')
print(soup.get_text())
```

The `get_text()` method called on the `soup` object will fetch us the HTML stripped content of the file. If you run the code written so far, you will get this output:

```
Full text HTML Stripped:
Sample Web Page

Main heading
This is a very simple HTML document
Improve your image by including an image.
Add a link to your favorite Web site.
This is a new sentence without a paragraph break, in bold italics.
This is purely the contents of our sample HTML document without any
of the HTML tags.
```

4. Sometimes, it's not enough to have pure HTML stripped content. You may also need specific tag contents. Let's access one of the tags:

```
print('Accessing the <title> tag :', end=' ')
print(soup.title)
```

The `soup.title` will return the first title tag it encounters in the file. Output of these lines will look like:

```
Accessing the <title> tag : <title>Sample Web Page</title>
```

5. Let us get only the HTML stripped text from a tag now. We will grab the text of the `<h1>` tag with the following piece of code:

```
print('Accessing the text of <H1> tag :', end=' ')
print(soup.h1.string)
```

The command `soup.h1.string` will return the text surrounded by the first `<h1>` tag encountered. The output of this line will be:

```
Accessing the text of <H1> tag : Main heading
```

6. Now we will access attributes of a tag. In this case, we will access the `alt` attribute of the `img` tag; add the following lines of code:

```
print('Accessing property of <img> tag :', end=' ')
print(soup.img['alt'])
```

Look carefully; the syntax to access attributes of a tag is different than accessing the text. When you run this piece of code, you will get this output:

```
Accessing property of <img> tag : A Great HTML Resource
```

7. Finally, there can be multiple occurrences of any type of tag in an HTML file. Simply using the `.` syntax will only fetch you the first instance. To fetch all instances, we use the `find_all()` functionality, shown as follows:

```
print('\nAccessing all occurrences of the <p> tag :')
for p in soup.find_all('p'):
    print(p.string)
```

The `find_all()` function called on a `BeautifulSoup` object will take as an argument the name of the tag, search through the entire HTML tree, and return all occurrences of that tag as a list. We are accessing that list in the `for` loop and printing the content/text of all the `<p>` tags in the given `BeautifulSoup` object:

Output: Full text HTML Stripped:

Sample Web Page

Main heading

This is a very simple HTML document

Improve your image by including an image.

Add a link to your favorite Web site.

This is a new sentence without a paragraph break, in bold italics.

Accessing the <title> tag : <title>Sample Web Page</title>

Accessing the text of <H1> tag : Main heading

Accessing property of tag : A Great HTML Resource

Accessing all occurrences of the <p> tag :

This is a very simple HTML document

Improve your image by including an image.

None

How it works...

BeautifulSoup 4 is a very handy library used to parse any HTML and XML content. It supports Python's inbuilt HTML parser, but you can also use other third-party parsers with it, for example, the `lxml` parser and the pure-Python `html5lib` parser. In this recipe, we used the Python inbuilt HTML parser. The output generated is pretty much self-explanatory, and of course, the assumption is that you do know what HTML is and how to write simple HTML.

3

Pre-Processing

In this chapter, we will cover the following recipes:

- Tokenization – learning to use the inbuilt tokenizers of NLTK
- Stemming – learning to use the inbuilt stemmers of NLTK
- Lemmatization – learning to use the WordnetLemmatizer of NLTK
- Stopwords – learning to use the stopwords corpus and seeing the difference it can make
- Edit distance – writing your own algorithm to find edit distance between two strings
- Processing two short stories and extracting the common vocabulary between two of them

Introduction

In the previous chapter, we learned to read, normalize, and organize raw data coming from heterogeneous forms and formats into uniformity. In this chapter, we will go a step forward and prepare the data for consumption in our NLP applications. Preprocessing is the most important step in any kind of data processing task, or else we fall prey to the age old computer science cliché of *garbage in, garbage out*. The aim of this chapter is to introduce some of the critical preprocessing steps such as tokenization, stemming, lemmatization, and so on.

In this chapter, we will be seeing six different recipes. We will build up the chapter by performing each preprocessing task in individual recipes—tokenization, stemming, lemmatization, stopwords treatment, and edit distance—in that order. In the last recipe, we will look at an example of how we can combine some of these preprocessing techniques to find common vocabulary between two free-form texts.

Tokenization – learning to use the inbuilt tokenizers of NLTK

Understand the meaning of tokenization, why we need it, and how to do it.

Getting ready

Let's first see what a token is. When you receive a document or a long string that you want to process or work on, the first thing you'd want to do is break it into words and punctuation marks. This is what we call the process of tokenization. We will see what types of tokenizers are available with NLTK and implement them as well.

How to do it...

1. Create a file named `tokenizer.py` and add the following import lines to it:

```
from nltk.tokenize import LineTokenizer, SpaceTokenizer,
TweetTokenizer
from nltk import word_tokenize
```

Import the four different types of tokenizers that we are going to explore in this recipe.

2. We will start with `LineTokenizer`. Add the following two lines:

```
lTokenizer = LineTokenizer();
print("Line tokenizer output :",lTokenizer.tokenize("My name is
Maximus Decimus Meridius, commander of the Armies of the North,
General of the Felix Legions and loyal servant to the true emperor,
Marcus Aurelius. \nFather to a murdered son, husband to a murdered
wife. \nAnd I will have my vengeance, in this life or the next."))
```

3. As the name implies, this tokenizer is supposed to divide the input string into lines (not sentences, mind you!). Let's see the output and what the tokenizer does:

```
Line tokenizer output : ['My name is Maximus Decimus Meridius,
commander of the Armies of the North, General of the Felix Legions
and loyal servant to the true emperor, Marcus Aurelius. ', 'Father
to a murdered son, husband to a murdered wife. ', 'And I will have
my vengeance, in this life or the next.']
```

As you can see, it has returned a list of three strings, meaning the given input has been divided in to three lines on the basis of where the newlines are.

`LineTokenizer` simply divides the given input string into new lines.

4. Now we will see `SpaceTokenizer`. As the name implies, it is supposed to divide on split on space characters. Add the following lines:

```
rawText = "By 11 o'clock on Sunday, the doctor shall open the
dispensary."
sTokenizer = SpaceTokenizer()
print("Space Tokenizer output :", sTokenizer.tokenize(rawText))
```

5. The `sTokenizer` object is an object of `SpaceTokenizer`. We have invoked the `tokenize()` method and we shall see the output now:

```
Space Tokenizer output : ['By', '11', "o'clock", 'on', 'Sunday',
'the', 'doctor', 'shall', 'open', 'the', 'dispensary.']
```

6. As expected, the input `rawText` is split on the space character `" "`. On to the next one! It's the `word_tokenize()` method. Add the following line:

```
print("Word Tokenizer output :", word_tokenize(rawText))
```

7. See the difference here. The other two we have seen so far are classes, whereas this is a method of the `nltk` module. This is the method that we will be using most of the time going forward as it does exactly what we've defined to be tokenization. It breaks up words and punctuation marks. Let's see the output:

```
Word Tokenizer output : ['By', '11', "o'clock", 'on', 'Sunday',
',', 'the', 'doctor', 'shall', 'open', 'the', 'dispensary', '.']
```

8. As you can see, the difference between `SpaceTokenizer` and `word_tokenize()` is clearly visible.

9. Now, on to the last one. There's a special `TweetTokernizer` that we can use when dealing with special case strings:

```
tTokenizer = TweetTokenizer()
print("Tweet Tokenizer output :",tTokenizer.tokenize("This is a
coool #dummysmiley: :-) :-P <3"))
```

10. Tweets contain special words, special characters, hashtags, and smileys that we want to keep intact. Let's see the output of these two lines:

```
Tweet Tokenizer output : ['This', 'is', 'a', 'coool',
'#dummysmiley', ':', ':-)', ':-P', '<3']
```

As we see, the `Tokenizer` kept the hashtag word intact and didn't break it; the smileys are also kept intact and are not lost. This is one special little class that can be used when the application demands it.

11. Here's the output of the program in full. We have already seen it in detail, so I will not be going into it again:

```
Line tokenizer output : ['My name is Maximus Decimus Meridius,
commander of the Armies of the North, General of the Felix Legions
and loyal servant to the true emperor, Marcus Aurelius. ', 'Father
to a murdered son, husband to a murdered wife. ', 'And I will have
my vengeance, in this life or the next.']
Space Tokenizer output : ['By', '11', "o'clock", 'on', 'Sunday,',
'the', 'doctor', 'shall', 'open', 'the', 'dispensary.']
Word Tokenizer output : ['By', '11', "o'clock", 'on', 'Sunday',
',', 'the', 'doctor', 'shall', 'open', 'the', 'dispensary', '.']
Tweet Tokenizer output : ['This', 'is', 'a', 'coool',
'#dummysmiley', ':', ':-)', ':-P', '<3']
```

How it works...

We saw three tokenizer classes and a method implemented to do the job in the `NLTK` module. It's not very difficult to understand how to do it, but it is worth knowing why we do it. The smallest unit to process in language processing task is a token. It is very much like a divide-and-conquer strategy, where we try to make sense of the smallest units at a granular level and add them up to understand the semantics of the sentence, paragraph, document, and the corpus (if any) by moving up the level of detail.

Stemming – learning to use the inbuilt stemmers of NLTK

Let's understand the concept of a stem and the process of stemming. We will learn why we need to do it and how to perform it using inbuilt NLTK stemming classes.

Getting ready

So what is a stem supposed to be? A stem is the base form of a word without any suffixes. And a stemmer is what removes the suffixes and returns the stem of the word. Let's see what types of stemmers are available with NLTK.

How to do it...

1. Create a file named `stemmers.py` and add the following import lines to it:

```
from nltk import PorterStemmer, LancasterStemmer, word_tokenize
```

Importing the four different types of tokenizers that we are going to explore in this recipe

2. Before we apply any stems, we need to tokenize the input text. Let's quickly get that done with the following code:

```
raw = "My name is Maximus Decimus Meridius, commander of the Armies  
of the North, General of the Felix Legions and loyal servant to the  
true emperor, Marcus Aurelius. Father to a murdered son, husband to  
a murdered wife. And I will have my vengeance, in this life or the  
next."  
tokens = word_tokenize(raw)
```

The token list contains all the tokens generated from the raw input string.

3. First we shall see `PorterStemmer`. Let's add the following three lines:

```
porter = PorterStemmer()  
pStems = [porter.stem(t) for t in tokens]  
print(pStems)
```


4. First, we initialize the stemmer object. Then we apply the stemmer to all tokens of the input text, and finally we print the output. Let's see the output and we will know more:

```
[ 'My', 'name', 'is', 'maximu', 'decimu', 'meridiu', ',', 'command',
  'of', 'the', 'armi', 'of', 'the', 'north', ',', 'gener', 'of',
  'the', 'felix', 'legion', 'and', 'loyal', 'servant', 'to', 'the',
  'true', 'emperor', ',', 'marcu', 'aureliu', '.', 'father', 'to',
  'a', 'murder', 'son', ',', 'husband', 'to', 'a', 'murder', 'wife',
  '.', 'and', 'I', 'will', 'have', 'my', 'vengeanc', ',', 'in',
  'thi', 'life', 'or', 'the', 'next', '.']
```

As you can see in the output, all the words have been rid of the trailing 's', 'es', 'e', 'ed', 'al', and so on.

5. The next one is `LancasterStemmer`. This one is supposed to be even more error prone as it contains many more suffixes to be removed than `porter`:

```
lancaster = LancasterStemmer()
lStems = [lancaster.stem(t) for t in tokens]
print(lStems)
```

6. The same drill! Just that this time we have `LancasterStemmer` instead of `PrterStemmer`. Let's see the output:

```
[ 'my', 'nam', 'is', 'maxim', 'decim', 'meridi', ',', 'command',
  'of', 'the', 'army', 'of', 'the', 'nor', ',', 'gen', 'of', 'the',
  'felix', 'leg', 'and', 'loy', 'serv', 'to', 'the', 'tru', 'emp',
  ',', 'marc', 'aureli', '.', 'fath', 'to', 'a', 'murd', 'son', ',',
  'husband', 'to', 'a', 'murd', 'wif', '.', 'and', 'i', 'wil', 'hav',
  'my', 'veng', ',', 'in', 'thi', 'lif', 'or', 'the', 'next', '.']
```

We shall discuss the difference in the output section, but we can make out that the suffixes that are dropped are bigger than Porter. 'us', 'e', 'th', 'eral', "ered", and many more!

7. Here's the output of the program in full. We will compare the output of both the stemmers:

```
[ 'My', 'name', 'is', 'maximu', 'decimu', 'meridiu', ',', 'command',
  'of', 'the', 'armi', 'of', 'the', 'north', ',', 'gener', 'of',
  'the', 'felix', 'legion', 'and', 'loyal', 'servant', 'to', 'the',
  'true', 'emperor', ',', 'marcu', 'aureliu', '.', 'father', 'to',
  'a', 'murder', 'son', ',', 'husband', 'to', 'a', 'murder', 'wife',
  '.', 'and', 'I', 'will', 'have', 'my', 'vengeanc', ',', 'in',
  'thi', 'life', 'or', 'the', 'next', '.']
```

```
['my', 'nam', 'is', 'maxim', 'decim', 'meridi', ',', 'command',
'of', 'the', 'army', 'of', 'the', 'nor', ',', 'gen', 'of', 'the',
'felix', 'leg', 'and', 'loy', 'serv', 'to', 'the', 'tru', 'emp',
',', 'marc', 'aureli', '.', 'fath', 'to', 'a', 'murd', 'son', ',',
'husband', 'to', 'a', 'murd', 'wif', '.', 'and', 'i', 'wil', 'hav',
'my', 'veng', ',', 'in', 'thi', 'lif', 'or', 'the', 'next', '.']
```

As we compare the output of both the stemmers, we see that `lancaster` is clearly the greedier one when dropping suffixes. It tries to remove as many characters from the end as possible, whereas `porter` is non-greedy and removes as little as possible.

How it works...

For some language processing tasks, we ignore the form available in the input text and work with the stems instead. For example, when you search on the Internet for *cameras*, the result includes documents containing the word *camera* as well as *cameras*, and vice versa. In hindsight though, both words are the same; the stem is *camera*.

Having said this, we can clearly see that this method is quite error prone, as the spellings are quite meddled with after a stemmer is done reducing the words. At times, it might be okay, but if you really want to understand the semantics, there is a lot of data loss here. For this reason, we shall next see what is called **lemmatization**.

Lemmatization – learning to use the WordnetLemmatizer of NLTK

Understand what lemma and lemmatization are. Learn how lemmatization differs from Stemming, why we need it, and how to perform it using `nltk` library's

`WordnetLemmatizer`.

Getting ready

A lemma is a lexicon headword or, more simply, the base form of a word. We have already seen what a stem is, but a lemma is a dictionary-matched base form unlike the stem obtained by removing/replacing the suffixes. Since it is a dictionary match, lemmatization is a slower process than Stemming.

How to do it...

1. Create a file named `lemmatizer.py` and add the following import lines to it:

```
from nltk import word_tokenize, PorterStemmer, WordNetLemmatizer
```

We will need to tokenize the sentences first, and we shall use the `PorterStemmer` to compare the output.

2. Before we apply any stems, we need to tokenize the input text. Let's quickly get that done with the following code:

```
raw = "My name is Maximus Decimus Meridius, commander of the armies  
of the north, General of the Felix legions and loyal servant to the  
true emperor, Marcus Aurelius. Father to a murdered son, husband to  
a murdered wife. And I will have my vengeance, in this life or the  
next."  
tokens = word_tokenize(raw)
```

The token list contains all the tokens generated from the `raw` input string.

3. First we will apply `PorterStemmer`, which we have already seen in the previous recipe. Let's add the following three lines:

```
porter = PorterStemmer()  
stems = [porter.stem(t) for t in tokens]  
print(stems)
```

First, we initialize the stemmer object. Then we apply the stemmer on all `tokens` of the input text, and finally we print the output. We shall check the output at the end of the recipe.

4. Now we apply the `lemmatizer`. Add the following three lines:

```
lemmatizer = WordNetLemmatizer()  
lemmas = [lemmatizer.lemmatize(t) for t in tokens]  
print(lemmas)
```

5. Run, and the output of these three lines will be like this:

```
['My', 'name', 'is', 'Maximus', 'Decimus', 'Meridius', ',', ',',
'commander', 'of', 'the', 'army', 'of', 'the', 'north', ',', ',',
'General', 'of', 'the', 'Felix', 'legion', 'and', 'loyal',
'servant', 'to', 'the', 'true', 'emperor', ',', ',', 'Marcus',
'Aurelius', '.', 'Father', 'to', 'a', 'murdered', 'son', ',', ',',
'husband', 'to', 'a', 'murdered', 'wife', '.', 'And', 'I', 'will',
'have', 'my', 'vengeance', ',', ',', 'in', 'this', 'life', 'or', 'the',
'next', '.']
```

As you see, it understands that for nouns it doesn't have to remove the trailing 's'. But for non-nouns, for example, legions and armies, it removes suffixes and also replaces them. However, what it's essentially doing is a dictionary match. We shall discuss the difference in the output section.

6. Here's the output of the program in full. We will compare the output of both the stemmers:

```
['My', 'name', 'is', 'maximu', 'decimu', 'meridiu', ',', ',', 'command',
'of', 'the', 'armi', 'of', 'the', 'north', ',', ',', 'gener', 'of',
'the', 'felix', 'legion', 'and', 'loyal', 'servant', 'to', 'the',
'true', 'emperor', ',', ',', 'marcu', 'aureliu', '.', 'father', 'to',
'a', 'murder', 'son', ',', ',', 'husband', 'to', 'a', 'murder', 'wife',
 '.', 'and', 'I', 'will', 'have', 'my', 'vengeanc', ',', ',', 'in',
'thi', 'life', 'or', 'the', 'next', '.']
['My', 'name', 'is', 'Maximus', 'Decimus', 'Meridius', ',', ',',
'commander', 'of', 'the', 'army', 'of', 'the', 'north', ',', ',',
'General', 'of', 'the', 'Felix', 'legion', 'and', 'loyal',
'servant', 'to', 'the', 'true', 'emperor', ',', ',', 'Marcus',
'Aurelius', '.', 'Father', 'to', 'a', 'murdered', 'son', ',', ',',
'husband', 'to', 'a', 'murdered', 'wife', '.', 'And', 'I', 'will',
'have', 'my', 'vengeance', ',', ',', 'in', 'this', 'life', 'or', 'the',
'next', '.']
```

As we compare the output of the stemmer and the lemmatizer, we see that the stemmer makes a lot of mistakes and the lemmatizer makes very few mistakes. However, it doesn't do anything with the word 'murdered', and that is an error. Yet, as an end product, lemmatizer does a far better job of getting us the base form than the stemmer.

How it works...

`WordNetLemmatizer` removes affixes only if it can find the resulting word in the dictionary. This makes the process of lemmatization slower than Stemming. Also, it understands and treats capitalized words as special words; it doesn't do any processing for them and returns them as is. To work around this, you may want to convert your input string to lowercase and then run lemmatization on it.

All said and done, lemmatization is still not perfect and will make mistakes. Check the input string and the result of this recipe; it couldn't convert 'murdered' to 'murder'. Similarly, it will handle the word 'women' correctly but can't handle 'men'.

Stopwords – learning to use the stopwords corpus and seeing the difference it can make

We will be using the Gutenberg corpus as an example in this recipe. The Gutenberg corpus is part of the NLTK data module. It contains a selection of 18 texts from some 25,000 electronic books from the project Gutenberg text archives. It is `PlainTextCorpus`, meaning there are no categories involved with this corpus. It is best suited if you want to play around with the words/tokens without worrying about the affinity of the text to any particular topic. One of the objectives of this little recipe is also to introduce one of the most important preprocessing steps in text analytics—stopwords treatment.

In accordance with the objectives, we will use this corpus to elaborate the usage of Frequency Distribution of the NLTK module in Python within the context of stopwords. To give a small synopsis, a stopword is a word that, though it has significant syntactic value in sentence formation, carries very negligible or minimal semantic value. When you are not working with the syntax but with a bag-of-words kind of approach (for example, TF/IDF), it makes sense to get rid of stopwords except the ones that you are specifically interested in.

Getting ready

The `nltk.corpus.stopwords` is also a corpus as part of the NLTK Data module that we will use in this recipe, along with `nltk.corpus.gutenberg`.

How to do it...

1. Create a new file named `Gutenberg.py` and add the following three lines of code to it:

```
import nltk
from nltk.corpus import gutenberg
print(gutenberg.fileids())
```

2. Here we are importing the required libraries and the Gutenberg corpus in the first two lines. The second line is used to check if the corpus was loaded successfully. Run the file on the Python interpreter and you should get an output that looks similar to:

```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt',
'bible-kjv.txt', 'blake-poems.txt', 'bryant-stories.txt', 'burgess-
busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt',
'chesterton-brown.txt', 'chesterton-thursday.txt', 'edgeworth-
parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt',
'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-
macbeth.txt', 'whitman-leaves.txt']
```

As you can see, the names of all 18 Gutenberg texts are printed on the console.

3. Add the following two lines of code, where we are doing a little preprocessing step on the list of all words from the corpus:

```
gb_words = gutenberg.words('bible-kjv.txt')
words_filtered = [e for e in gb_words if len(e) >= 3]
```

The first line simply copies the list of all words in the corpus from the sample `bible—kjv.txt` in the `gb_words` variable. The second, and interesting, step is where we are iterating over the entire list of words from Gutenberg, discarding all the words/tokens whose length is two characters or less.

4. Now we will access `nltk.corpus.stopwords` and do stopwords treatment on the filtered words list from the previous list. Add the following lines of code for the same:

```
stopwords = nltk.corpus.stopwords.words('english')
words = [w for w in words_filtered if w.lower() not in stopwords]
```

The first line simply loads words from the stopwords corpus into the `stopwords` variable for the english language. The second line is where we are filtering out all stopwords from the filtered word list we had developed in the previous example.

- Now we will simply apply `nltk.FreqDist` to the list of preprocessed words and the plain list of words. Add these lines to do the same:

```
fdistPlain = nltk.FreqDist(words)
fdist = nltk.FreqDist(gb_words)
```

Create the `FreqDist` object by passing as argument the words list that we formulated in steps 2 and 3.

- Now we want to see some of the characteristics of the frequency distribution that we just made. Add the following four lines in the code and we will see what each does:

```
print('Following are the most common 10 words in the bag')
print(fdistPlain.most_common(10))
print('Following are the most common 10 words in the bag minus the
stopwords')
print(fdist.most_common(10))
```

The `most_common(10)` function will return the 10 most common words in the word bag being processed by frequency distribution. What it outputs is what we will discuss and elaborate now.

- After you run this program, you should get something similar to the following:

```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt',
'bible-kjv.txt', 'blake-poems.txt', 'bryant-stories.txt', 'burgess-
busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt',
'chesterton-brown.txt', 'chesterton-thursday.txt', 'edgeworth-
parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt',
'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-
macbeth.txt', 'whitman-leaves.txt']
```

Following are the most common 10 words in the bag

```
[(',', 70509), ('the', 62103), (':', 43766), ('and', 38847), ('of',
34480), ('.', 26160), ('to', 13396), ('And', 12846), ('that',
12576), ('in', 12331)]
```

Following are the most common 10 words in the bag minus the stopwords

```
[('shall', 9838), ('unto', 8997), ('lord', 7964), ('thou', 5474),
```

```
('thy', 4600), ('god', 4472), ('said', 3999), ('thee', 3827),  
('upon', 2748), ('man', 2735)]
```

How it works...

If you look carefully at the output, the most common 10 words in the unprocessed or plain list of words won't make much sense. Whereas from the preprocessed bag of words, the most common 10 words such as `god`, `lord`, and `man` give us a quick understanding that we are dealing with a text related to faith or religion.

The foremost objective of this recipe is to introduce you to the concept of stopwords treatment for text preprocessing techniques that you would most likely have to do before running any complex analysis on your data. The NLTK stopwords corpus contains stopwords for 11 languages. When you are trying to analyze the importance of keywords in any text analytics application, treating the stopwords properly will take you a long way. Frequency distribution will help you get the importance of words. Statistically speaking, this distribution would ideally look like a bell curve if you plot it on a two-dimensional plane of frequency and importance of words.

Edit distance – writing your own algorithm to find edit distance between two strings

Edit distance, also called as **Levenshtein distance** is a metric used to measure the similarity between two distances. Essentially, it's a count of how many edit operations, deletions, insertions, or substitutions will transform a given String A to String B. We shall write our own algorithm to calculate the edit distance and then compare it against `nltk.metrics.distance.edit_distance()` for a sanity check.

Getting ready

You may want to look up a little more on the Levenshtein distance part for mathematical equations. We will look at the algorithm implementation in python and why we do it, but it may not be feasible to cover the complete mathematics behind it. Here's a link on Wikipedia: https://en.wikipedia.org/wiki/Levenshtein_distance.

How to do it...

1. Create a file named `edit_distance_calculator.py` and add the following import lines to it:

```
from nltk.metrics.distance import edit_distance
```

We just imported the inbuilt `nltk` library's `edit_distance` function from the `nltk.metrics.distance` module.

2. Let's define our method to accept two strings and calculate the edit distance between the two. `str1` and `str2` are two strings that the function accepts, and we will return an integer distance value:

```
def my_edit_distance(str1, str2):
```

3. The next step is to get the length of the two input strings. We will be using the `length` to create an $m \times n$ table where `m` and `n` are the lengths of the two strings `s1` and `s2` respectively:

```
m=len(str1)+1
n=len(str2)+1
```

4. Now we will create `table` and initialize the first row and first column:

```
table = {}
for i in range(m): table[i,0]=i
for j in range(n): table[0,j]=j
```

5. This will initialize the two-dimensional array and the contents will look like the following table in memory:

	0	1	2	3	4	
0	0	1	2	3	4	
1	1					A
2	2					N
3	3					D
		H	A	N	D	

Please note that this is inside a function and I'm using the example strings we are going to pass to the function to elaborate the algorithm.

6. Now comes the tricky part. We are going to fill up the matrix using the formula:

```
for i in range(1, m):
    for j in range(1, n):
        cost = 0 if str1[i-1] == str2[j-1] else 1
        table[i,j] = min(table[i, j-1]+1, table[i-1, j]+1, table[i-1,
j-1]+cost)
```

The cost is calculated on whether the characters in contention are the same or they edition, specifically deletion or insertion. The formula in the next line for is calculating the value of the cell in the matrix, the first two take care of substitution and the third one is for substitution. We also add the cost of the previous step to it and take the minimum of the three.

7. At the end, we return the value of the last cell, that is, `table[m,n]`, as the final edit distance value:

```
return table[i,j]
```

8. Now we will call our function and the `nltk` library's `edit_distance()` function on two strings and check the output:

```
print("Our Algorithm :",my_edit_distance("hand", "and"))
print("NLTK Algorithm :",edit_distance("hand", "and"))
```

9. Our words are hand and and. Only a single delete operation on the first string or a single insertion operation on the second string will give us a match. Hence, the expected Levenshtein score is 1.
10. Here's the output of the program:

```
Our Algorithm : 1
NLTK Algorithm : 1
```

As expected, the `NLTK edit_distance()` returns 1 and so does our algorithm. Fair to say that our algorithm is doing as expected, but I would urge you guys to test it further by running it through with some more examples.

How it works...

I've already given you a brief on the algorithm; now let's see how the matrix *table* gets populated with the algorithm. See the attached table here:

	0	1	2	3	4	
0	0	1	2	3	4	
1	1	1	1	2	3	A
2	2	2	2	1	2	N
3	3	3	3	2	1	D
		H	A	N	D	

You've already seen how we initialized the matrix. Then we filled up the matrix using the formula in algorithm. The yellow trail you see is the significant numbers. After the first iteration, you can see that the distance is moving in the direction of 1 consistently and the final value that we return is denoted by the green background cell.

Now, the applications of the edit distance algorithm are multifold. First and foremost, it is used in spell checkers and auto-suggestions in text editors, search engines, and many such text-based applications. Since the cost of comparisons is equivalent to the product of the length of the strings to be compared, it is sometimes impractical to apply it to compare large texts.

Processing two short stories and extracting the common vocabulary between two of them

This recipe is supposed to give you an idea of how to handle a typical text analytics problem when you come across it. We will be using multiple preprocessing techniques in the process of getting to our outcome. The recipe will end with an important preprocessing task and not a real application of text analysis. We will be using a couple of short stories from <http://www.english-for-students.com/>.

Getting ready

We will be removing all special characters, splitting words, doing case folds, and some set and list operations in this recipe. We won't be using any special libraries, just Python programming tricks.

How to do it...

1. Create a file named `lemmatizer.py` and create a couple of long strings with short stories or any news articles:

```
story1 = """In a far away kingdom, there was a river. This river
was home to many golden swans. The swans spent most of their time
on the banks of the river. Every six months, the swans would leave
a golden feather as a fee for using the lake. The soldiers of the
kingdom would collect the feathers and deposit them in the royal
treasury.
One day, a homeless bird saw the river. "The water in this river
seems so cool and soothing. I will make my home here," thought the
bird.
As soon as the bird settled down near the river, the golden swans
noticed her. They came shouting. "This river belongs to us. We pay
a golden feather to the King to use this river. You can not live
here."
"I am homeless, brothers. I too will pay the rent. Please give me
shelter," the bird pleaded. "How will you pay the rent? You do not
have golden feathers," said the swans laughing. They further added,
"Stop dreaming and leave once." The humble bird pleaded many times.
But the arrogant swans drove the bird away.
"I will teach them a lesson!" decided the humiliated bird.
She went to the King and said, "O King! The swans in your river are
impolite and unkind. I begged for shelter but they said that they
had purchased the river with golden feathers."
The King was angry with the arrogant swans for having insulted the
homeless bird. He ordered his soldiers to bring the arrogant swans
to his court. In no time, all the golden swans were brought to the
King's court.
"Do you think the royal treasury depends upon your golden feathers?
You can not decide who lives by the river. Leave the river at once
or you all will be beheaded!" shouted the King.
The swans shivered with fear on hearing the King. They flew away
never to return. The bird built her home near the river and lived
there happily forever. The bird gave shelter to all other birds in
the river. """
story2 = """Long time ago, there lived a King. He was lazy and
```

liked all the comforts of life. He never carried out his duties as a King. "Our King does not take care of our needs. He also ignores the affairs of his kingdom." The people complained. One day, the King went into the forest to hunt. After having wandered for quite sometime, he became thirsty. To his relief, he spotted a lake. As he was drinking water, he suddenly saw a golden swan come out of the lake and perch on a stone. "Oh! A golden swan. I must capture it," thought the King. But as soon as he held his bow up, the swan disappeared. And the King heard a voice, "I am the Golden Swan. If you want to capture me, you must come to heaven." Surprised, the King said, "Please show me the way to heaven." "Do good deeds, serve your people and the messenger from heaven would come to fetch you to heaven," replied the voice. The selfish King, eager to capture the Swan, tried doing some good deeds in his Kingdom. "Now, I suppose a messenger will come to take me to heaven," he thought. But, no messenger came. The King then disguised himself and went out into the street. There he tried helping an old man. But the old man became angry and said, "You need not try to help. I am in this miserable state because of out selfish King. He has done nothing for his people." Suddenly, the King heard the golden swan's voice, "Do good deeds and you will come to heaven." It dawned on the King that by doing selfish acts, he will not go to heaven. He realized that his people needed him and carrying out his duties was the only way to heaven. After that day he became a responsible King.

"""

There we have two short stories from the aforementioned website!

2. First, we will remove some of the special characters from the texts. We are removing all newlines (`'\n'`), commas, full stops, exclamations, question marks, and so on. At the end, we convert the entire string to lowercase with the `casefold()` function:

```
story1 = story1.replace(", ", "").replace("\n", "").replace('.', ' '),
''.replace('!', '').replace("?", "").casefold()
story2 = story2.replace(", ", "").replace("\n", "").replace('.', ' '),
''.replace('!', '').replace("?", "").casefold()
```

3. Next, we will split the texts into words:

```
story1_words = story1.split(" ")
print("First Story words :", story1_words)
story2_words = story2.split(" ")
print("Second Story words :", story2_words)
```

4. Using `split` on the `" "` character, we split and get the list of words from `story1` and `story2`. Let's see the output after this step:

```
First Story words : ['in', 'a', 'far', 'away', 'kingdom', 'there',
'was', 'a', 'river', 'this', 'river', 'was', 'home', 'to', 'many',
'golden', 'swans', 'the', 'swans', 'spent', 'most', 'of', 'their',
'time', 'on', 'the', 'banks', 'of', 'the', 'river', 'every', 'six',
'months', 'the', 'swans', 'would', 'leave', 'a', 'golden',
'feather', 'as', 'a', 'fee', 'for', 'using', 'the', 'lake', 'the',
'soldiers', 'of', 'the', 'kingdom', 'would', 'collect', 'the',
'feathers', 'and', 'deposit', 'them', 'in', 'the', 'royal',
'treasury', 'one', 'day', 'a', 'homeless', 'bird', 'saw', 'the',
'river', 'the', 'water', 'in', 'this', 'river', 'seems', 'so',
'cool', 'and', 'soothing', 'i', 'will', 'make', 'my', 'home',
'here', 'thought', 'the', 'bird', 'as', 'soon', 'as', 'the',
'bird', 'settled', 'down', 'near', 'the', 'river', 'the', 'golden',
'swans', 'noticed', 'her', 'they', 'came', 'shouting', 'this',
'river', 'belongs', 'to', 'us', 'we', 'pay', 'a', 'golden',
'feather', 'to', 'the', 'king', 'to', 'use', 'this', 'river',
'you', 'can', 'not', 'live', 'here', 'i', 'am', 'homeless',
'brothers', 'i', 'too', 'will', 'pay', 'the', 'rent', 'please',
'give', 'me', 'shelter', 'the', 'bird', 'pleaded', 'how', 'will',
'you', 'pay', 'the', 'rent', 'you', 'do', 'not', 'have', 'golden',
'feathers', 'said', 'the', 'swans', 'laughing', 'they', 'further',
'added', 'stop', 'dreaming', 'and', 'leave', 'once', 'the',
'humble', 'bird', 'pleaded', 'many', 'times', 'but', 'the',
'arrogant', 'swans', 'drove', 'the', 'bird', 'away', 'i', 'will',
'teach', 'them', 'a', 'lesson', 'decided', 'the', 'humiliated',
'bird', 'she', 'went', 'to', 'the', 'king', 'and', 'said', 'o',
'king', 'the', 'swans', 'in', 'your', 'river', 'are', 'impolite',
'and', 'unkind', 'i', 'begged', 'for', 'shelter', 'but', 'they',
'said', 'that', 'they', 'had', 'purchased', 'the', 'river', 'with',
'golden', 'feathers', 'the', 'king', 'was', 'angry', 'with', 'the',
'arrogant', 'swans', 'for', 'having', 'insulted', 'the',
'homeless', 'bird', 'he', 'ordered', 'his', 'soldiers', 'to',
'bring', 'the', 'arrogant', 'swans', 'to', 'his', 'court', 'in',
'no', 'time', 'all', 'the', 'golden', 'swans', 'were', 'brought',
'to', 'the', 'king's', 'court', 'do', 'you', 'think', 'the',
'royal', 'treasury', 'depends', 'upon', 'your', 'golden',
'feathers', 'you', 'can', 'not', 'decide', 'who', 'lives', 'by',
'the', 'river', 'leave', 'the', 'river', 'at', 'once', 'or', 'you',
'all', 'will', 'be', 'beheaded', 'shouted', 'the', 'king', 'the',
'swans', 'shivered', 'with', 'fear', 'on', 'hearing', 'the',
'king', 'they', 'flew', 'away', 'never', 'to', 'return', 'the',
'bird', 'built', 'her', 'home', 'near', 'the', 'river', 'and',
'lived', 'there', 'happily', 'forever', 'the', 'bird', 'gave',
'shelter', 'to', 'all', 'other', 'birds', 'in', 'the', 'river', '']
Second Story words : ['long', 'time', 'ago', 'there', 'lived', 'a',
```

```
'king', 'he', 'was', 'lazy', 'and', 'liked', 'all', 'the',
'comforts', 'of', 'life', 'he', 'never', 'carried', 'out', 'his',
'duties', 'as', 'a', 'king', "our", 'king', 'does', 'not', 'take',
'care', 'of', 'our', 'needs', 'he', 'also', 'ignores', 'the',
'affairs', 'of', 'his', 'kingdom', 'the', 'people', 'complained',
'one', 'day', 'the', 'king', 'went', 'into', 'the', 'forest', 'to',
'hunt', 'after', 'having', 'wandered', 'for', 'quite', 'sometime',
'he', 'became', 'thirsty', 'to', 'his', 'relief', 'he', 'spotted',
'a', 'lake', 'as', 'he', 'was', 'drinking', 'water', 'he',
'suddenly', 'saw', 'a', 'golden', 'swan', 'come', 'out', 'of',
'the', 'lake', 'and', 'perch', 'on', 'a', 'stone', "oh", 'a',
'golden', 'swan', 'i', 'must', 'capture', 'it', 'thought', 'the',
'king', 'but', 'as', 'soon', 'as', 'he', 'held', 'his', 'bow',
'up', 'the', 'swan', 'disappeared', 'and', 'the', 'king', 'heard',
'a', 'voice', "i", 'am', 'the', 'golden', 'swan', 'if', 'you',
'want', 'to', 'capture', 'me', 'you', 'must', 'come', 'to',
'heaven', 'surprised', 'the', 'king', 'said', "please", 'show',
'me', 'the', 'way', 'to', 'heaven', "do", 'good', 'deeds',
'serve', 'your', 'people', 'and', 'the', 'messenger', 'from',
'heaven', 'would', 'come', 'to', 'fetch', 'you', 'to', 'heaven',
'replied', 'the', 'voice', 'the', 'selfish', 'king', 'eager', 'to',
'capture', 'the', 'swan', 'tried', 'doing', 'some', 'good',
'deeds', 'in', 'his', 'kingdom', "now", 'i', 'suppose', 'a',
'messenger', 'will', 'come', 'to', 'take', 'me', 'to', 'heaven',
'he', 'thought', 'but', 'no', 'messenger', 'came', 'the', 'king',
'then', 'disguised', 'himself', 'and', 'went', 'out', 'into',
'the', 'street', 'there', 'he', 'tried', 'helping', 'an', 'old',
'man', 'but', 'the', 'old', 'man', 'became', 'angry', 'and',
'said', "you", 'need', 'not', 'try', 'to', 'help', 'i', 'am',
'in', 'this', 'miserable', 'state', 'because', 'of', 'out',
'selfish', 'king', 'he', 'has', 'done', 'nothing', 'for', 'his',
'people', 'suddenly', 'the', 'king', 'heard', 'the', 'golden',
'swan's', 'voice', "do", 'good', 'deeds', 'and', 'you', 'will',
'come', 'to', 'heaven', 'it', 'dawned', 'on', 'the', 'king',
'that', 'by', 'doing', 'selfish', 'acts', 'he', 'will', 'not',
'go', 'to', 'heaven', 'he', 'realized', 'that', 'his', 'people',
'needed', 'him', 'and', 'carrying', 'out', 'his', 'duties', 'was',
'the', 'only', 'way', 'to', 'heaven', 'after', 'that', 'day', 'he',
'became', 'a', 'responsible', 'king', '']
```

As you can see, all the special characters are gone and a list of words is created.

5. Now let's create a vocabulary out of this list of words. A vocabulary is a set of words. No repeats!

```
story1_vocab = set(story1_words)
print("First Story vocabulary :", story1_vocab)
story2_vocab = set(story2_words)
```

```
print("Second Story vocabulary",story2_vocab)
```

6. Calling the Python internal `set()` function on the list will deduplicate the list and convert it into a set:

```
First Story vocabulary : {'', 'king's', 'am', 'further', 'having',
'river', 'he', 'all', 'feathers', 'banks', 'at', 'shivered',
'other', 'are', 'came', 'here', 'that', 'soon', 'lives', 'unkind',
'by', 'on', 'too', 'kingdom', 'never', 'o', 'make', 'every',
'will', 'said', 'birds', 'teach', 'away', 'hearing', 'humble',
'but', 'deposit', 'them', 'would', 'leave', 'return', 'added',
'were', 'fear', 'bird', 'lake', 'my', 'settled', 'or', 'pleaded',
'in', 'so', 'use', 'was', 'me', 'us', 'laughing', 'bring', 'rent',
'have', 'how', 'lived', 'of', 'seems', 'gave', 'day', 'no',
'months', 'down', 'this', 'the', 'her', 'decided', 'angry',
'built', 'cool', 'think', 'golden', 'spent', 'time', 'noticed',
'lesson', 'many', 'near', 'once', 'collect', 'who', 'your', 'flew',
'fee', 'six', 'most', 'had', 'to', 'please', 'purchased',
'happily', 'depends', 'belongs', 'give', 'begged', 'there', 'she',
'i', 'times', 'dreaming', 'as', 'court', 'their', 'you', 'shouted',
'shelter', 'forever', 'royal', 'insulted', 'they', 'with', 'live',
'far', 'water', 'king', 'shouting', 'a', 'brothers', 'drove',
'arrogant', 'saw', 'soldiers', 'stop', 'home', 'upon', 'can',
'decide', 'beheaded', 'do', 'for', 'homeless', 'ordered', 'be',
'using', 'not', 'feather', 'soothing', 'swans', 'humiliated',
'treasury', 'thought', 'one', 'and', 'we', 'impolite', 'brought',
'went', 'pay', 'his'}

Second Story vocabulary {'', 'needed', 'having', 'am', 'he', 'all',
'way', 'spotted', 'voice', 'realized', 'also', 'came', 'that',
'our', 'soon', 'oh', 'by', 'on', 'has', 'complained', 'never',
'ago', 'kingdom', 'do', 'capture', 'said', 'into', 'long', 'will',
'liked', 'disappeared', 'but', 'would', 'must', 'stone', 'lake',
'from', 'messenger', 'eager', 'deeds', 'fetch', 'carrying', 'in',
'because', 'perch', 'responsible', 'was', 'me', 'disguised',
'take', 'comforts', 'lived', 'of', 'tried', 'day', 'no', 'street',
'good', 'bow', 'the', 'need', 'this', 'helping', 'angry', 'out',
'thirsty', 'relief', 'wandered', 'old', 'golden', 'acts', 'time',
'an', 'needs', 'suddenly', 'state', 'serve', 'affairs', 'ignores',
'does', 'people', 'want', 'your', 'dawned', 'man', 'to',
'miserable', 'became', 'swan', 'there', 'hunt', 'show', 'i',
'heaven', 'as', 'selfish', 'after', 'suppose', 'you', 'only',
'done', 'drinking', 'then', 'care', 'it', 'him', 'come', 'swan's',
'if', 'water', 'himself', 'nothing', 'please', 'carried', 'king',
'help', 'heard', 'up', 'try', 'a', 'held', 'saw', 'life',
'surprised', 'go', 'i', 'for', 'doing', 'our', 'some', 'now',
'sometime', 'forest', 'lazy', 'not', 'you', 'replied', 'quite',
'duties', 'thought', 'one', 'and', 'went', 'his'}
```


Here are the deduplicated sets, the vocabularies of both the stories.

7. Now, the final step. Produce the common vocabulary between these two stories:

```
common_vocab = story1_vocab & story2_vocab  
print("Common Vocabulary :",common_vocab)
```

8. Python allows the set operation & (AND), which we are using to find the set of common entries between these two vocabulary sets. Let's see the output of the final step:

```
Common Vocabulary : {'', 'king', 'am', 'having', 'he', 'all',  
'your', 'in', 'was', 'me', 'a', 'to', 'came', 'that', 'lived',  
'soon', 'saw', 'of', 'by', 'on', 'day', 'no', 'never', 'kingdom',  
'there', 'for', 'i', 'said', 'will', 'the', 'this', 'as', 'angry',  
'you', 'not', 'but', 'would', 'golden', 'thought', 'time', 'one',  
'and', 'lake', 'went', 'water', 'his'}
```

And there it is, the end-goal.

Here is the output:

```
I won't be dumping the output of the entire program again here. It's huge  
so let's save some trees!
```

How it works...

So here, we saw how we can go from a couple of narratives to the common vocabulary between them. We didn't use any fancy libraries, nor did we perform any complex operations. Yet we built a base from which we can take this bag-of-words forward and do many things with it.

From here on, we can think of many different applications, such as text similarity, search engine tagging, text summarization, and many more.

4

Regular Expressions

In this chapter, we will cover the following recipes:

- Regular expression – learning to use *, +, and ?
- Regular expression – learning to use \$ and ^, and the non-start and non-end of a word
- Searching multiple literal strings and substring occurrence
- Learning to create date regex and a set of characters or ranges of character
- Finding all five character words and making abbreviations in some sentences
- Learning to write your own regex tokenizer
- Learning to write your own regex stemmer

Introduction

In the previous chapter, we saw what preprocessing tasks you would want to perform on your raw data. This chapter, immediately after, provides an excellent opportunity to introduce regular expressions. Regular expressions are one of the most simple and basic, yet most important and powerful, tools that you will learn. More commonly known as regex, they are used to match patterns in text. We will learn exactly how powerful this is in this chapter.

We do not claim that you will be an expert in writing regular expressions after this chapter and that is perhaps not the goal of this book or this chapter. The aim of this chapter is to introduce you to the concept of pattern matching as a way to do text analysis and for this, there is no better tool to start with than regex. By the time you finish the recipes, you shall feel fairly confident of performing any text match, text split, text search, or text extraction operation.

Let's look at the aforementioned recipes in detail.

Regular expression – learning to use *, +, and ?

We start off with a recipe that will elaborate the use of the `*`, `+`, and `?` operators in regular expressions. These short-hand operators are more commonly known as wild cards, but I prefer to call them zero or more (`*`) one or more (`+`), and zero or one (`?`) for distinction. These names are much more intuitive if you think about them.

Getting ready

The regular expressions library is a part of the Python package and no additional packages need to be installed.

How to do it...

1. Create a file named `regex1.py` and add the following `import` line to it:

```
import re
```

This imports the `re` object, which allows processing and implementation of regular expressions.

2. Add the following Python function in the file that is supposed to apply the given patterns for matching:

```
def text_match(text, patterns):
```

This function accepts two arguments; `text` is the input text on which the `patterns` will be applied for match.

3. Now, let's define the function. Add the following lines under the function:

```
    if re.search(patterns, text):  
        return 'Found a match!'  
    else:  
        return('Not matched!')
```

The `re.search()` method applies the given pattern to the `text` object and returns `true` or `false` depending on the outcome after applying the method. That is the end of our function.

4. Let's apply the wild card patterns one by one. We start with zero or one:

```
print(text_match("ac", "ab?"))
print(text_match("abc", "ab?"))
print(text_match("abbc", "ab?"))
```

5. Let's look at this pattern `ab?`. What this means is a followed by zero or one `b`. Let's see what the output will be when we execute these three lines:

```
Found a match!
Found a match!
Found a match!
```

Now, all of them found a match. These patterns are trying to match a part of the input and not the entire input; hence, they find a match with all three inputs.

6. On to the next one, zero or more! Add the following three lines:

```
print(text_match("ac", "ab*"))
print(text_match("abc", "ab*"))
print(text_match("abbc", "ab*"))
```

7. The same set of inputs but a different string. The pattern says, a followed by zero or more `b`. Let's see the output of these three lines:

```
Found a match!
Found a match!
Found a match!
```

As you can see, all the texts find a match. As rule of thumb, whatever matches zero or one wild card will also match zero or more. The `?` wildcard is a subset of `*`.

8. Now, the one or more wild card. Add the following lines:

```
print(text_match("ac", "ab+"))
print(text_match("abc", "ab+"))
print(text_match("abbc", "ab+"))
```

9. The same input! Just that the pattern contains the + one or more wild card. Let's see the output:

```
Not matched!  
Found a match!  
Found a match!
```

As you can see, the first input string couldn't find the match. The rest did as expected.

10. Now, being more specific in the number of repetitions, add the following line:

```
print(text_match("abbc", "ab{2}"))
```

The pattern says a followed by exactly two b. Needless to say, the pattern will find a match in the input text.

11. Time for a range of repetitions! Add the following line:

```
print(text_match("aabbbbc", "ab{3,5}?"))
```

This will also be a match as we have as a substring a followed by four b.

The output of the program won't really make much sense in full. We have already `ana.y`sed the output of each and every step; hence, we won't be printing it down here again.

How it works...

The `re.search()` function is a function that will only apply the given pattern as a test and will return true or false as the result of the test. It won't return the matching value. For that, there are other `re` functions that we shall learn in later recipes.

Regular expression – learning to use \$ and ^, and the non-start and non-end of a word

The starts with (^) and ends with (\$) operators are indicators used to match the given patterns at the start or end of an input text.

Getting ready

We could have reused the `text_match()` function from the previous recipe, but instead of importing an external file, we shall rewrite it. Let's look at the recipe implementation.

How to do it...

1. Create a file named `regex2.py` and add the following `import` line to it:

```
import re
```

2. Add this Python function in the file that is supposed to apply the given patterns for matching:

```
def text_match(text, patterns):
    if re.search(patterns, text):
        return 'Found a match!'
    else:
        return('Not matched!')
```

This function accepts two arguments; `text` is the input text on which `patterns` will be applied for matching and will return whether the match was found or not. The function is exactly what we wrote in the previous recipe.

3. Let's apply the following pattern. We start with a simple starts with ends with:

```
print("Pattern to test starts and ends with")
print(text_match("abbc", "^a.*c$"))
```

4. Let's look at this pattern, `^a.*c$`. This means: start with `a`, followed by zero or more of any characters, and end with `c`. Let's see the output when we execute these three lines:

```
Pattern to test starts and ends with

Found a match!
```

It found a match for the input text, of course. What we introduced here is a new `.` wildcard. The dot matches any character except a newline in default mode; that is, when you say `.*`, it means zero or more occurrences of any character.

5. On to the next one, to find a pattern that looks for an input text that begins with a word. Add the following two lines:

```
print("Begin with a word")
print(text_match("Tuffy eats pie, Loki eats peas!", "^\\w+"))
```

6. `\\w` stands for any alphanumeric character and underscore. The pattern says: start with (^) any alphanumeric character (\\w) and one or more occurrences of it (+). The output:

```
Begin with a word

Found a match!
```

As expected, the pattern finds a match.

7. Next, we check for an ends with a word and optional punctuation. Add the following lines:

```
print("End with a word and optional punctuation")
print(text_match("Tuffy eats pie, Loki eats peas!", "\\w+\\S*?$"))
```

8. The pattern means one or more occurrences of `\\w`, followed by zero or more occurrences of `\\S`, and that should be falling towards the end of the input text. To understand `\\S` (capital S), we must first understand `\\s`, which is all whitespace characters. `\\S` is the reverse or the anti-set of `\\s`, which when followed by `\\w` translates to looking for a punctuation:

```
End with a word and optional punctuation

Found a match!
```

We found the match with peas! at the end of the input text.

9. Next, find a word that contains a specific character. Add the following lines:

```
print("Finding a word which contains character, not start or end of
the word")
print(text_match("Tuffy eats pie, Loki eats peas!", "\\Bu\\B"))
```

For decoding this pattern, `\B` is a anti-set or reverse of `\b`. The `\b` matches an empty string at the beginning or end of a word, and we have already seen what a word is. Hence, `\B` will match inside the word and it will match any word in our input string that contains character `u`:

```
Finding a word which contains character, not start or end of the
word
```

```
Found a match!
```

We find the match in the first word, *Tuffy*.

Here's the output of the program in full. We have already seen it in detail, so I will not go into it again:

```
Pattern to test starts and ends with
```

```
Found a match!
```

```
Begin with a word
```

```
Found a match!
```

```
End with a word and optional punctuation
```

```
Found a match!
```

```
Finding a word which contains character, not start or end of the word
```

```
Found a match!
```

How it works...

Along with starts with and ends with, we also learned the wild card character `.` and some other special sequences such as `\w`, `\s`, `\b`, and so on.

Searching multiple literal strings and substring occurrences

In this recipe, we shall run some iterative functions with regular expressions. More specifically, we shall run multiple patterns on an input string with a `for` loop and we shall also run a single pattern for multiple matches on the input. Let's directly see how to do it.

Getting ready

Open your PyCharm editor or any other Python editor that you use, and you are ready to go.

How to do it...

1. Create a file named `regex3.py` and add the following `import` line to it:

```
import re
```

2. Add the following two Python lines to declare and define our patterns and the input text:

```
patterns = [ 'Tuffy', 'Pie', 'Loki' ]  
text = 'Tuffy eats pie, Loki eats peas!'
```

3. Let us write our first `for` loop. Add these lines:

```
for pattern in patterns:  
    print('Searching for "%s" in "%s" -&gt;' % (pattern, text),)  
    if re.search(pattern, text):  
        print('Found!')  
    else:  
        print('Not Found!')
```

This is a simple `for` loop, iterating on the list of patterns one by one and calling the search function of `re`. Run this piece and you shall find a match for two of the three words in the input string. Also, do note that these patterns are case sensitive; the capitalized word `Tuffy`! We will discuss the output in the output section.

4. On to the next one, to search a substring and find its location too. Let's define the pattern and the input text first:

```
text = 'Diwali is a festival of lights, Holi is a festival of  
colors!'  
pattern = 'festival'
```

The preceding two lines define the input text and the pattern to search for respectively.

5. Now, the `for` loop that will iterate over the input text and fetch all occurrences of the given pattern:

```
for match in re.finditer(pattern, text):  
    s = match.start()  
    e = match.end()  
    print('Found "%s" at %d:%d' % (text[s:e], s, e))
```

6. The `finditer` function takes as input the pattern and the input text on which to apply that pattern. On the returned list, we shall iterate. For every object, we will call the `start` and `end` methods to know the exact location where we found a match for the pattern. We will discuss the output of this block here. The output of this little block will look like:

```
Found "festival" at 12:20
```

```
Found "festival" at 42:50
```

Two lines of output! Which suggests that we found the pattern at two places in the input. The first was at position 12:20 and the second was at 42:50 as displayed in the output text lines.

Here's the output of the program in full. We have already seen some parts in detail but we will go through it again:

```
Searching for "Tuffy" in "Tuffy eats pie, Loki eats peas!" -&gt;
```

```
Found!
```

```
Searching for "Pie" in "Tuffy eats pie, Loki eats peas!" -&gt;
```

```
Not Found!
```

```
Searching for "Loki" in "Tuffy eats pie, Loki eats peas!" -&gt;
```

```
Found!
```

```
Found "festival" at 12:20
```

```
Found "festival" at 42:50
```

The output is quite intuitive, or at least the first six lines are. We searched for the word `Tuffy` and it was found. The word `Pie` wasn't found (the `re.search()` function is case sensitive) and then the word `Loki` was found. The last two lines we've already discussed, in the sixth step. We didn't just search the string but also pointed out the index where we found them in the given input.

How it works...

Let's discuss some more things about the `re.search()` function we have used quite heavily so far. As you can see in the preceding output, the word `pie` is part of the input text but we search for the capitalized word `Pie` and we can't seem to locate it. If you add a flag in the search function call `re.IGNORECASE`, only then will it be a case-insensitive search. The syntax will be `re.search(pattern, string, flags=re.IGNORECASE)`.

Now, the `re.finditer()` function. The syntax of the function is `re.finditer(pattern, string, flags=0)`. It returns an iterator containing `MatchObject` instances over all the non-overlapping matches found in the input string.

Learning to create date regex and a set of characters or ranges of character

In this recipe, we shall first run a simple date regex. Along with that, we will learn the significance of the `()` groups. Since that's too less to include in a recipe, we shall also throw in some more things like the squared brackets `[]`, which indicate a set (we will see in detail what a set is).

How to do it...

1. Create a file named `regex4.py` and add the following `import` line to it:

```
import re
```

2. Let's declare a `url` object and write a simple date finder regular expression to start:

```
url=
"http://www.telegraph.co.uk/formula-1/2017/10/28/mexican-grand-prix-2017-time-does-start-tv-channel-odds-lewis1/"

date_regex = '(/(\d{4})/(\d{1,2})/(\d{1,2})/'
```

The `url` is a simple string object. The `date_regex` is also a simple string object but it contains a regex that will match a date with format `YYYY/DD/MM` or `YYYY/MM/DD` type of dates. `\d` denotes digits starting from 0 to 9. We've already learned the notation `{}`.

3. Let's apply `date_regex` to `url` and see the output. Add the following line:

```
print("Date found in the URL :", re.findall(date_regex, url))
```

4. A new `re` function, `re.findall(pattern, input, flags=0)`, which again accepts the pattern, the input text, and optionally flags (we learned case sensitive flag in the previous recipe). Let's see the output:

```
Date found in the URL : [('2017', '10', '28')]
```

So, we've found the date 28 October 2017 in the given input string object.

5. Now comes the next part, where we will learn about the set of characters notation `[]`. Add the following function in the code:

```
def is_allowed_specific_char(string):
    charRe = re.compile(r'^a-zA-Z0-9.~')
    string = charRe.search(string)
    return not bool(string)
```

The purpose here is to check whether the input string contains a specific set of characters or others. Here, we are going with a slightly different approach; first, we `re.compile` the pattern, which returns a `RegexObject`. Then, we call the `search` method of `RegexObject` on the already compiled pattern. If a match is found, the `search` method returns a `MatchObject`, and `None` otherwise. Now, turning our attention to the set notation `[]`. The pattern enclosed inside the squared brackets means: not (`^`) the range of characters `a-z`, `A-Z`, `0-9`, or `..`. Effectively, this is an OR operation of all things enclosed by the squared brackets.

6. Now the test for the pattern. Let's call the function on two different types of inputs, one that matches and one that doesn't:

```
print(is_allowed_specific_char("ABCDEFabcdef123450."))
print(is_allowed_specific_char("*&%@#!}{") )
```

7. The first set of characters contains all of the allowed list of characters, whereas the second set contains all of the disallowed set of characters. As expected, the output of these two lines will be:

True

False

The pattern will iterate through each and every character of the input string and see if there is any disallowed character, and it will flag it out. You can try adding any of the disallowed set of characters in the first call of `is_allwoed_specific_char()` and check for yourself.

Here's the output of the program in full. We have already seen it in detail, so we shall not go through it again:

Date found in the URL : [('2017', '10', '28')]

True

False

How it works...

Let's first discuss what a group is. A group in any regular expression is what is enclosed inside the brackets `()` inside the pattern declaration. If you see the output of the date match, you will see a set notation, inside which you have three string objects: `[('2017', '10', '28')]`. Now, look at the pattern declared carefully, `/(\d{4})/(\d{1,2})/(\d{1,2})/`. All the three components of the date are marked inside the group notation `()`, and hence all three are identified separately.

Now, the `re.findall()` method will find all the matches in the given input. This means that if there were more dates inside the give input text, the output would've looked like `[('2017', '10', '28'), ('2015', '05', '12')]`.

The `[]` notation that is set essentially means: match either of the characters enclosed inside the set notation. If any single match is found, the pattern is true.

Find all five-character words and make abbreviations in some sentences

We have covered all the important notations that I wanted to cover with examples in the previous recipes. Now, going forward, we will look at a few small recipes that are geared more towards accomplishing a certain task using regular expressions than explaining any notations. Needless to say, we will still learn some more notations.

How to do it...

1. Create a file named `regex_assignment1.py` and add the following `import` line to it:

```
import re
```

2. Add the following two Python lines to define the input string and apply the substitution pattern for abbreviation:

```
street = '21 Ramkrishna Road'
print(re.sub('Road', 'Rd', street))
```

3. First, we are going to do the abbreviation, for which we use the `re.sub()` method. The pattern to look for is `Road`, the string to replace it with `Rd`, and the input is the string object `street`. Let's look at the output:

```
21 Ramkrishna Rd
```

Clearly, it works as expected.

4. Now, let us find all five-character words inside any given sentence. Add these two lines of code for that:

```
text = 'Diwali is a festival of light, Holi is a festival of color!'
print(re.findall(r"\b\w{5}\b", text))
```

5. Declare a string object `text` and put the sentence side it. Next, create a pattern and apply it using the `re.findall()` function. We are using the `\b` boundary set to identify the boundary between words and the `{}` notation to make sure we are only shortlisting five-character words. Run this and you shall see the list of words matched as expected:

```
['light', 'color']
```

Here's the output of the program in full. We have already seen it in detail, so we will not go through it again:

```
21 Ramkrishna Rd  
['light', 'color']
```

How it works...

By now, I assume you have a good understanding of the regular expression notations and syntax. Hence, the explanations given when we wrote the recipe are quite enough. Instead, let us look at something more interesting. Look at the `findall()` method; you will see a notation like `r<lt;pattern>gt;`. This is called the raw string notation; it helps keep the regular expression sane looking. If you don't do it, you will have to provide an escape sequence to all the backslashes in your regular expression. For example, patterns `r"\b\w{5}\b"` and `"\\b\\w{5}\\b"` do the exact same job functionality wise.

Learning to write your own regex tokenizer

We already know the concepts of tokens, tokenizers, and why we need them from the previous chapter. We have also seen how to use the inbuilt tokenizers of the NLTK module. In this recipe, we will write our own tokenizer; it will evolve to mimic the behavior of `nltk.word_tokenize()`.

Getting ready

If you have your Python interpreter and editor ready, you are as ready as you can ever be.

How to do it...

1. Create a file named `regex_tokenizer.py` and add the following `import` line to it:

```
import re
```

2. Let's define our raw sentence to tokenize and the first pattern:

```
raw = "I am big! It's the pictures that got small."

print(re.split(r' ', raw))
```

3. This pattern will perform the same as the space tokenizer we saw in previous chapter. Let's look at the output:

```
['I', 'am', 'big!', "It's", 'the', 'pictures', 'that', 'got',
'small.']
```

As we can see, our little pattern works exactly as expected.

4. Now, this is not enough, is it? We want to split the tokens on anything non-word and not the ' ' characters alone. Let's try the following pattern:

```
print(re.split(r'\W+', raw))
```

5. We are splitting on all non-word characters, that is, `\W`. Let's see the output:

```
['I', 'am', 'big', 'It', 's', 'the', 'pictures', 'that', 'got',
'small', '']
```

We did split out on all the non-word characters (' ', ,, !, and so on), but we seem to have removed them from the result altogether. Looks like we need to do something more and different.

6. Split doesn't seem to be doing the job; let's try a different `re` function, `re.findall()`. Add the following line:

```
print(re.findall(r'\w+|\S\w*', raw))
```

7. Let's run and see the output:

```
['I', 'am', 'big', '!', 'It', "'s", 'the', 'pictures', 'that',
'got', 'small', '.']
```


Looks like we hit the jackpot.

Here's the output of the program in full. We have already discussed it; let's print it out:

```
['I', 'am', 'big!', "It's", 'the', 'pictures', 'that', 'got', 'small.']  
  
['I', 'am', 'big', 'It', 's', 'the', 'pictures', 'that', 'got', 'small',  
 '']  
  
['I', 'am', 'big', '!', 'It', "'s", 'the', 'pictures', 'that', 'got',  
 'small', '.']
```

As you can see, we have gradually improved upon our pattern and approach to achieve the best possible outcome in the end.

How it works...

We started with a simple `re.split` on space characters and improvised it using the non-word character. Finally, we changed our approach; instead of trying to split, we went about matching what we wanted by using `re.findall`, which did the job.

Learning to write your own regex stemmer

We already know the concept of stems/lemmas, stemmer, and why we need them from the previous chapter. We have seen how to use the inbuilt porter stemmer and Lancaster stemmer of the NLTK module. In this recipe, we will write our own regular expression stemmer that will get rid of the trailing unwanted suffixes to find the correct stems.

Getting ready

As we did in previous stemmer and lemmatizer recipes, we will need to tokenize the text before we apply the stemmer. That's exactly what we are going to do. We will reuse the final tokenizer pattern from the last recipe. If you haven't checked out the previous recipe, please do so and you are ready set to start this one.

How to do it...

1. Create a file named `regex_tokenizer.py` and add the following `import` line to it:

```
import re
```

2. We will write a function that will do the job of stemming for us. Let's first declare the syntax of the function in this step and we will define it in the next step:

```
def stem(word):
```

This function shall accept a string object as parameter and is supposed to return a string object as the outcome. Word in stem out!

3. Let's define the `stem()` function:

```
splits = re.findall(r'^(.*?) (ing|ly|ed|ious|ies|ive|es|s|ment)?$',  
word)  
stem = splits[0][0]  
return stem
```

We are applying the `re.findall()` function to the input word to return two groups as output. First is the stem and then it's any possible suffix. We return the first group as our result from the function call.

4. Let's define our input sentence and tokenize it. Add the following lines:

```
raw = "Keep your friends close, but your enemies closer."  
tokens = re.findall(r'\w+|\S\w*', raw)  
print(tokens)
```

5. Let's run and see the output:

```
['Keep', 'your', 'friends', 'close', ',', 'but', 'your', 'enemies',  
'closer', '.']
```

Looks like we got our tokens to do stemming.

6. Let's apply our `stem()` method to the list of tokens we just generated. Add the following `for` loop:

```
for t in tokens:
    print("'" + stem(t) + "'")
```

We are just looping over all tokens and printing the returned stem one by one. We will see the output in the upcoming output section and discuss it there.

Let's see the output of the entire code:

```
['Keep', 'your', 'friends', 'close', ',', 'but', 'your', 'enemies',
'closer', '.']

'Keep'

'your'

'friend'

'close'

','

'but'

'your'

'enem'

'closer'

'.'
```

Our stemmer seems to be doing a pretty decent job. However, I reckon I have passed an easy-looking sentence for the stemmer.

How it works...

Again, we are using the `re.findall()` function to get the desired output, though you might want to look closely at the first group's regex pattern. We are using a non-greedy wildcard match (`. *?`); otherwise, it will greedily gobble up the entire word and there will be no suffixes identified. Also, the start and end of the input are mandatory to match the entire input word and split it.

5

POS Tagging and Grammars

In this chapter, we will cover the following recipes:

- Exploring the in-built tagger
- Writing your own tagger
- Training your own tagger
- Learning to write your own grammar
- Writing a probabilistic context-free grammar--CFG
- Writing a recursive CFG

Introduction

This chapter primarily focuses on learning the following subjects using Python NLTK:

- Taggers
- CFG

Tagging is the process of classifying the words in a given sentence using **parts of speech (POS)**. Software that helps achieve this is called **tagger**. NLTK has support for a variety of taggers. We will go through the following taggers as part of this chapter:

- In-built tagger
- Default tagger
- Regular expression tagger
- Lookup tagger

CFG describes a set of rules that can be applied to text in a formal language specification to generate newer sets of text.

CFG in a language comprises the following things:

- Starting token
- A set of tokens that are terminals (ending symbols)
- A set of tokens that are non-terminals (non-ending symbols)
- Rules or productions that define rewrite rules that help transform non-terminals to either terminals or non-terminals

Exploring the in-built tagger

In the following recipe, we use the Python NLTK library to understand more about the POS tagging features in a given text.

We will make use of the following technologies from the Python NLTK library:

- Punkt English tokenizer
- Averaged perception tagger

The datasets for these taggers can be downloaded from your NLTK distribution by invoking `nltk.download()` from the Python prompt.

Getting ready

You should have a working Python (Python 3.6 is preferred) installed in your system along with the NLTK library and all its collections for optimal experience.

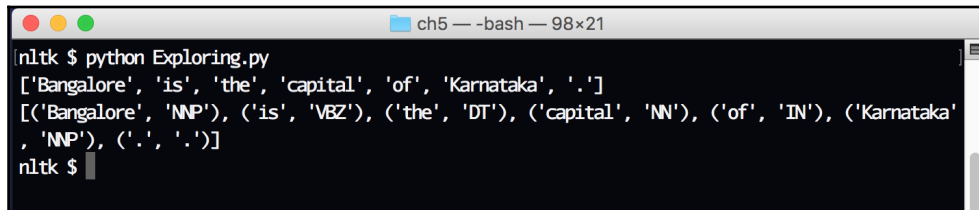
How to do it...

1. Open atom editor (or favorite programming editor).
2. Create a new file called `Exploring.py`.

3. Type the following source code:

```
Exploring.py
1 import nltk
2 simpleSentence = "Bangalore is the capital of Karnataka."
3 wordsInSentence = nltk.word_tokenize(simpleSentence)
4 print(wordsInSentence)
5 partsOfSpeechTags = nltk.pos_tag(wordsInSentence)
6 print(partsOfSpeechTags)
7
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```
ch5 — -bash — 98x21
nltk $ python Exploring.py
['Bangalore', 'is', 'the', 'capital', 'of', 'Karnataka', '.']
[('Bangalore', 'NNP'), ('is', 'VBZ'), ('the', 'DT'), ('capital', 'NN'), ('of', 'IN'), ('Karnataka', 'NNP'), ('.', '.')]
nltk $
```

How it works...

Now, let's go through the program that we have just written and dig into the details:

```
import nltk
```

This is the first instruction in our program, which instructs the Python interpreter to load the module from disk to memory and make the NLTK library available for use in the program:

```
simpleSentence = "Bangalore is the capital of Karnataka."
```

In this instruction, we are creating a variable called `simpleSentence` and assigning a hard coded string to it:

```
wordsInSentence = nltk.word_tokenize(simpleSentence)
```

In this instruction, we are invoking the NLTK built-in tokenizer function `word_tokenize()`; it breaks a given sentence into words and returns a Python `list` datatype. Once the result is computed by the function, we assign it to a variable called `wordsInSentence` using the `=` (equal to) operator:

```
print(wordsInSentence)
```

In this instruction, we are calling the Python built-in `print()` function, which displays the given data structure on the screen. In our case, we are displaying the list of all words that are tokenized. See the output carefully; we are displaying a Python `list` data structure on screen, which consists of all the strings separated by commas, and all the list elements are enclosed in square brackets:

```
partsOfSpeechTags = nltk.pos_tag(wordsInSentence)
```

In this instruction we are invoking the NLTK built-in tagger `pos_tag()`, which takes a list of words in the `wordsInSentence` variable and identifies the POS. Once the identification is complete, a list of tuples. Each tuple has the tokenized word and the POS identifier:

```
print(partsOfSpeechTags)
```

In this instruction, we are invoking the Python built-in `print()` function, which prints the given parameter to the screen. In our case, we can see a list of tuples, where each tuple consists of the original word and POS identifier.

Writing your own tagger

In the following recipe, we will explore the NLTK library by writing our own taggers. The following types of taggers will be written:

- Default tagger
- Regular expression tagger
- Lookup tagger

Getting ready

You should have a working Python (Python 3.6 is preferred) installed in your system along with the NLTK library and all its collections for optimal experience.

You should also have `python-crfsuite` installed to run this recipe.

How to do it...

1. Open your atom editor (or favorite programming editor).
2. Create a new file called `OwnTagger.py`.
3. Type the following source code:

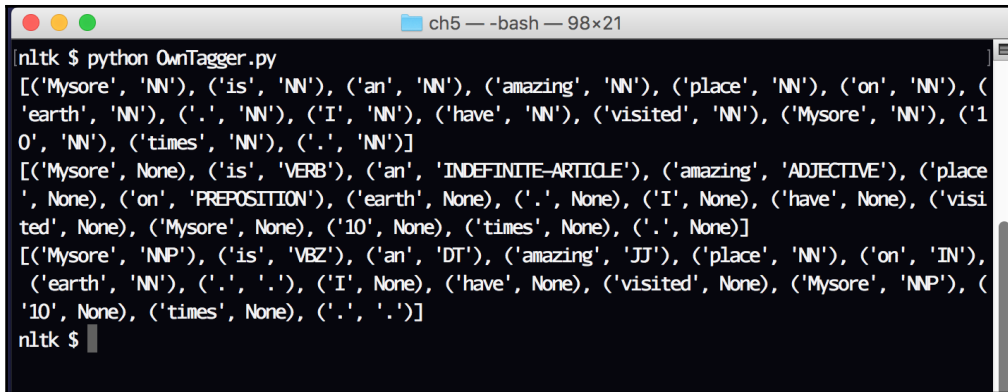
```

OwnTagger.py
1  import nltk
2  def learnDefaultTagger(simpleSentence):
3      wordsInSentence = nltk.word_tokenize(simpleSentence)
4      tagger = nltk.DefaultTagger("NN")
5      posEnabledTags = tagger.tag(wordsInSentence)
6      print(posEnabledTags)
7  def learnRETagger(simpleSentence):
8      customPatterns = [
9          (r'.*ing$', 'ADJECTIVE'),          # running
10         (r'.*ly$', 'ADVERB'),              # willingly
11         (r'.*ion$', 'NOUN'),               # intimation
12         (r'(. *ate|. *en|is)$', 'VERB'),    # terminate, darken, lighten
13         (r'^an$', 'INDEFINITE-ARTICLE'),   # terminate
14         (r'^(with|on|at)$', 'PREPOSITION'), # on
15         (r'^\d{1,3}(\.?\d{1,3})+$', 'NUMBER'), # -1.0, 12345.123
16         (r'.*$', None),
17     ]
18     tagger = nltk.RegexpTagger(customPatterns)
19     wordsInSentence = nltk.word_tokenize(simpleSentence)
20     posEnabledTags = tagger.tag(wordsInSentence)
21     print(posEnabledTags)
22  def learnLookupTagger(simpleSentence):
23     mapping = {
24         '.': '.', 'place': 'NN', 'on': 'IN',
25         'earth': 'NN', 'Mysore': 'NNP', 'is': 'VBZ',
26         'an': 'DT', 'amazing': 'JJ'
27     }
28     tagger = nltk.UnigramTagger(model=mapping)
29     wordsInSentence = nltk.word_tokenize(simpleSentence)
30     posEnabledTags = tagger.tag(wordsInSentence)
31     print(posEnabledTags)
32
33  if __name__ == '__main__':
34     testSentence = "Mysore is an amazing place on earth. I have visited Mysore 10 times."
35     learnDefaultTagger(testSentence)
36     learnRETagger(testSentence)
37     learnLookupTagger(testSentence)
38

```

4. Save the File.
5. Run the program using the Python interpreter.

6. You will see the following output.



```

ch5 — -bash — 98x21
nltk $ python OwnTagger.py
[('Mysore', 'NN'), ('is', 'NN'), ('an', 'NN'), ('amazing', 'NN'), ('place', 'NN'), ('on', 'NN'), ('earth', 'NN'), ('.', 'NN'), ('I', 'NN'), ('have', 'NN'), ('visited', 'NN'), ('Mysore', 'NN'), ('10', 'NN'), ('times', 'NN'), ('.', 'NN')]
[('Mysore', None), ('is', 'VERB'), ('an', 'INDEFINITE-ARTICLE'), ('amazing', 'ADJECTIVE'), ('place', None), ('on', 'PREPOSITION'), ('earth', None), ('.', None), ('I', None), ('have', None), ('visited', None), ('Mysore', None), ('10', None), ('times', None), ('.', None)]
[('Mysore', 'NNP'), ('is', 'VBZ'), ('an', 'DT'), ('amazing', 'JJ'), ('place', 'NN'), ('on', 'IN'), ('earth', 'NN'), ('.', '.'), ('I', None), ('have', None), ('visited', None), ('Mysore', 'NNP'), ('10', None), ('times', None), ('.', '.')]
nltk $

```

How it works...

Now, let's go through the program that we have just written to understand more:

```
import nltk
```

This is the first instruction in our program; it instructs the Python interpreter to load the module from disk to memory and make the NLTK library available for use in the program:

```
def learnDefaultTagger(simpleSentence):
    wordsInSentence = nltk.word_tokenize(simpleSentence)
    tagger = nltk.DefaultTagger("NN")
    posEnabledTags = tagger.tag(wordsInSentence)
    print(posEnabledTags)
```

All of these instructions are defining a new Python function that takes a string as input and prints the words in this sentence along with the default tag on screen. Let's further understand this function to see what it's trying to do:

```
def learnDefaultTagger(simpleSentence):
```

In this instruction, we are defining a new Python function called `learnDefaultTagger`; it takes a parameter named `simpleSentence`:

```
    wordsInSentence = nltk.word_tokenize(simpleSentence)
```

In this instruction, we are calling the `word_tokenize` function from the NLTK library. We are passing `simpleSentence` as the first parameter to this function. Once the data is computed by this function, the return value is stored in the `wordsInSentence` variable. Which are list of words:

```
tagger = nltk.DefaultTagger("NN")
```

In this instruction, we are creating an object of the `DefaultTagger()` class from the Python `nltk` library with `NN` as the argument passed to it. This will initialize the tagger and assign the instance to the `tagger` variable:

```
posEnabledTags = tagger.tag(wordsInSentence)
```

In this instruction, we are calling the `tag()` function of the `tagger` object, which takes the tokenized words from the `wordsInSentence` variable and returns the list of tagged words. This is saved in `posEnabledTags`. Remember that all the words in the sentence will be tagged as `NN` as that's what the tagger is supposed to do. This is like a very basic level of tagging without knowing anything about POS:

```
print(posEnabledTags)
```

Here we are calling Python's built-in `print()` function to inspect the contents of the `posEnabledTags` variable. We can see that all the words in the sentence will be tagged with `NN`:

```
def learnRETagger(simpleSentence):
    customPatterns = [
        (r'.*ing$', 'ADJECTIVE'),
        (r'.*ly$', 'ADVERB'),
        (r'.*ion$', 'NOUN'),
        (r'(*ate|*en|is)$', 'VERB'),
        (r'^an$', 'INDEFINITE-ARTICLE'),
        (r'^(with|on|at)$', 'PREPOSITION'),
        (r'^\d+([0-9]+(\.[0-9]+))$', 'NUMBER'),
        (r'.*$', None),
    ]
    tagger = nltk.RegexpTagger(customPatterns)
    wordsInSentence = nltk.word_tokenize(simpleSentence)
    posEnabledTags = tagger.tag(wordsInSentence)
    print(posEnabledTags)
```

These are the instructions to create a new function called `learnRETagger()`, which takes a string as input and prints the list of all tokens in the string with properly identified tags using the regular expression tagger as output.

Let's try to understand one instruction at a time:

```
def learnRETagger(simpleSentence):
```

We are defining a new Python function named *learnRETagger* to take a parameter called *simpleSentence*.

In order to understand the next instruction, we should learn more about Python lists, tuples, and regular expressions:

- A Python list is a data structure that is an ordered set of elements
- A Python tuple is a immutable (read-only) data structure that is an ordered set of elements
- Python regular expressions are strings that begin with the letter *r* and follow the standard PCRE notation:

```
customPatterns = [  
    (r'.*ing$', 'ADJECTIVE'),  
    (r'.*ly$', 'ADVERB'),  
    (r'.*ion$', 'NOUN'),  
    (r'(. *ate|. *en|is)$', 'VERB'),  
    (r'^an$', 'INDEFINITE-ARTICLE'),  
    (r'^(with|on|at)$', 'PREPOSITION'),  
    (r'^\d-?[0-9]+(\.\d[0-9]+)$', 'NUMBER'),  
    (r'.*$', None),  
]
```

Even though this looks big, this is a single instruction that does many things:

- Creating a variable called *customPatterns*
- Defining a new Python list datatype with *[*
- Adding eight elements to this list
- Each element in this list is a tuple that has two items in it
- The first item in the tuple is a regular expression
- The second item in the tuple is a string

Now, translating the preceding instruction into a human-readable form, we have added eight regular expressions to tag the words in a sentence to be any of ADJECTIVE, ADVERB, NOUN, VERB, INDEFINITE-ARTICLE, PREPOSITION, NUMBER, or None type.

We do this by identifying certain patterns in English words identifiable as a given POS.

In the preceding example, these are the clues we are using to tag the POS of English words:

- Words that end with `ing` can be called `ADJECTIVE`, for example, `running`
- Words that end with `ly` can be called `ADVERB`, for example, `willingly`
- Words that end with `ion` can be called `NOUN`, for example, `intimation`
- Words that end with `ate` or `en` can be called `VERB`, for example, `terminate`, `darken`, or `lighten`
- Words that end with `an` can be called `INDEFINITE-ARTICLE`
- Words such as `with`, `on`, or `at` are `PREPOSITION`
- Words that are like, `-123.0`, `984` can be called `NUMBER`
- We are tagging everything else as `None`, which is a built-in Python datatype used to represent nothing

```
tagger = nltk.RegexpTagger(customPatterns)
```

In this instruction, we are creating an instance of the NLTK built-in regular expression tagger `RegexpTagger`. We are passing the list of tuples in the `customPatterns` variable as the first parameter to the class to initialize the object. This object can be referenced in future with the variable named `tagger`:

```
wordsInSentence = nltk.word_tokenize(simpleSentence)
```

Following the general process, we first try to tokenize the string in `simpleSentence` using the NLTK built-in `word_tokenize()` function and store the list of tokens in the `wordsInSentence` variable:

```
posEnabledTags = tagger.tag(wordsInSentence)
```

Now we are invoking the regular expression tagger's `tag()` function to tag all the words that are in the `wordsInSentence` variable. The result of this tagging process is stored in the `posEnabledTags` variable:

```
print(posEnabledTags)
```

We are calling the Python built-in `print()` function to display the contents of the `posEnabledTags` data structure on screen:

```
def learnLookupTagger(simpleSentence):  
    mapping = {  
        ' ': '.', 'place': 'NN', 'on': 'IN',  
        'earth': 'NN', 'Mysore' : 'NNP', 'is': 'VBZ',
```

```
'an': 'DT', 'amazing': 'JJ'
}
tagger = nltk.UnigramTagger(model=mapping)
wordsInSentence = nltk.word_tokenize(simpleSentence)
posEnabledTags = tagger.tag(wordsInSentence)
print(posEnabledTags)
```

Let's take a closer look:

```
def learnLookupTagger(simpleSentence):
```

We are defining a new function, `learnLookupTagger`, which takes a string as parameter into the `simpleSentence` variable:

```
    tagger = nltk.UnigramTagger(model=mapping)
```

In this instruction, we are calling `UnigramTagger` from the `nltk` library. This is a lookup tagger that takes the Python dictionary we have created and assigned to the `mapping` variable. Once the object is created, it's available in the `tagger` variable for future use:

```
    wordsInSentence = nltk.word_tokenize(simpleSentence)
```

Here, we are tokenizing the sentence using the NLTK built-in `word_tokenize()` function and capturing the result in the `wordsInSentence` variable:

```
    posEnabledTags = tagger.tag(wordsInSentence)
```

Once the sentence is tokenized, we call the `tag()` function of the tagger by passing the list of tokens in the `wordsInSentence` variable. The result of this computation is assigned to the `posEnabledTags` variable:

```
    print(posEnabledTags)
```

In this instruction, we are printing the data structure in `posEnabledTags` on the screen for further inspection:

```
    testSentence = "Mysore is an amazing place on earth. I have visited Mysore  
    10 times."
```

We are creating a variable called `testSentence` and assigning a simple English sentence to it:

```
    learnDefaultTagger(testSentence)
```

We call the `learnDefaultTagger` function created in this recipe by passing the `testSentence` as the first argument to it. Once this function execution completes, we will see the sentence POS tagged:

```
learnRETagger(testSentence)
```

In this expression, we are invoking the `learnRETagger()` function with the same test sentence in the `testSentence` variable. The output from this function is a list of tags that are tagged as per the regular expressions that we have defined ourselves:

```
learnLookupTagger(testSentence)
```

The output from this function `learnLookupTagger` is list of all tags from the sentence `testSentence` that are tagged using the lookup dictionary that we have created.

Training your own tagger

In this recipe, we will learn how to train our own tagger and save the trained model to disk so that we can use it later for further computations.

Getting ready

You should have a working Python (Python 3.6 is preferred) installed in your system, along with the NLTK library and all its collections for optimal experience.

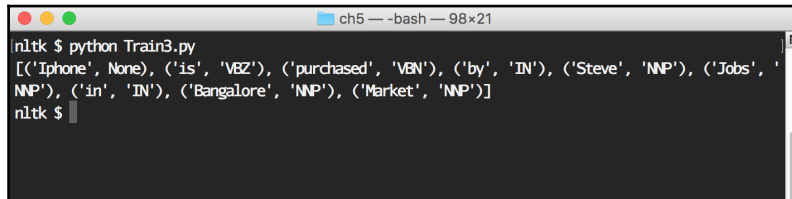
How to do it...

1. Open your atom editor (or favorite programming editor).
2. Create a new file called `Train3.py`.

3. Type the following source code:

```
Train3.py
1 import nltk
2 import pickle
3
4 def sampleData():
5     return [
6         "Bangalore is the capital of Karnataka.",
7         "Steve Jobs was the CEO of Apple.",
8         "iPhone was Invented by Apple.",
9         "Books can be purchased in Market.",
10    ]
11
12 def buildDictionary():
13     dictionary = {}
14     for sent in sampleData():
15         partsOfSpeechTags = nltk.pos_tag(nltk.word_tokenize(sent))
16         for tag in partsOfSpeechTags:
17             value = tag[0]
18             pos = tag[1]
19             dictionary[value] = pos
20     return dictionary
21
22 def saveMyTagger(tagger, fileName):
23     fileHandle = open(fileName, "wb")
24     pickle.dump(tagger, fileHandle)
25     fileHandle.close()
26
27 def saveMyTraining(fileName):
28     tagger = nltk.UnigramTagger(model=buildDictionary())
29     saveMyTagger(tagger, fileName)
30
31 def loadMyTagger(fileName):
32     return pickle.load(open(fileName, "rb"))
33
34 sentence = 'Iphone is purchased by Steve Jobs in Bangalore Market'
35 fileName = "myTagger.pickle"
36
37 saveMyTraining(fileName)
38
39 myTagger = loadMyTagger(fileName)
40
41 print(myTagger.tag(nltk.word_tokenize(sentence)))
42
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```
nltk $ python Train3.py
[('Iphone', None), ('is', 'VBZ'), ('purchased', 'VBN'), ('by', 'IN'), ('Steve', 'NNP'), ('Jobs', 'NNP'), ('in', 'IN'), ('Bangalore', 'NNP'), ('Market', 'NNP')]
nltk $
```

How it works...

Let's understand how the program works:

```
import nltk
import pickle
```

In these two instructions, we are loading the `nltk` and `pickle` modules into the program. The `pickle` module implements powerful serialization and de-serialization algorithms to handle very complex Python objects:

```
def sampleData():
    return [
        "Bangalore is the capital of Karnataka.",
        "Steve Jobs was the CEO of Apple.",
        "iPhone was Invented by Apple.",
        "Books can be purchased in Market.",
    ]
```

In these instructions, we are defining a function called `sampleData()` that returns a Python list. Basically, we are returning four sample strings:

```
def buildDictionary():
    dictionary = {}
    for sent in sampleData():
        partsOfSpeechTags = nltk.pos_tag(nltk.word_tokenize(sent))
        for tag in partsOfSpeechTags:
            value = tag[0]
            pos = tag[1]
            dictionary[value] = pos
    return dictionary
```


We now define a function called `buildDictionary()`; it reads one string at a time from the list generated by the `sampleData()` function. Each string is tokenized using the `nltk.word_tokenize()` function. The resultant tokens are added to a Python dictionary, where the dictionary key is the word in the sentence and the value is POS. Once a dictionary is computed, it's returned to the caller:

```
def saveMyTagger(tagger, fileName):
    fileHandle = open(fileName, "wb")
    pickle.dump(tagger, fileHandle)
    fileHandle.close()
```

In these instructions, we are defining a function called `saveMyTagger()` that takes two parameters:

- `tagger`: An object to the POS tagger
- `fileName`: This contains the name of the file to store the `tagger` object in

We first open the file in **write binary (wb)** mode. Then, using `pickle` module's `dump()` method, we store the entire `tagger` in the file and call the `close()` function on `fileHandle`:

```
def saveMyTraining(fileName):
    tagger = nltk.UnigramTagger(model=buildDictionary())
    saveMyTagger(tagger, fileName)
```

In these instructions, we are defining a new function called `saveMyTraining`; it takes a single argument called `fileName`.

We are building an `nltk.UnigramTagger()` object with the model as output from the `buildDictionary()` function (which itself is built from the sample set of strings that we have defined). Once the `tagger` object is created, we call the `saveMyTagger()` function to save it to disk:

```
def loadMyTagger(fileName):
    return pickle.load(open(fileName, "rb"))
```

Here, we are defining a new function, `loadMyTagger()`, which takes `fileName` as a single argument. This function reads the file from disk and passes it to the `pickle.load()` function which unserializes the `tagger` from disk and returns a reference to it:

```
sentence = 'Iphone is purchased by Steve Jobs in Bangalore Market'
fileName = "myTagger.pickle"
```

In these two instructions, we are defining two variables, `sentence` and `fileName`, which contain a sample string that we want to analyze and the file path at which we want to store the POS tagger respectively:

```
saveMyTraining(fileName)
```

This is the instruction that actually calls the function `saveMyTraining()` with `myTagger.pickle` as argument. So, we are basically storing the trained tagger in this file:

```
myTagger = loadMyTagger(fileName)
```

In this instruction, we take the `myTagger.pickle` as argument of the `loadMyTagger()` function, which loads the tagger from disk, deserializes it, and creates an object, which further gets assigned to the `myTagger` variable:

```
print(myTagger.tag(nltk.word_tokenize(sentence)))
```

In this instruction, we are calling the `tag()` function of the tagger that we have just loaded from disk. We use it to tokenize the sample string that we have created.

Once the processing is done, the output is displayed on the screen.

Learning to write your own grammar

In automata theory, CFG consists of the following things:

- A starting symbol/ token
- A set of symbols/ tokens that are terminals
- A set of symbols/ tokens that are non-terminals
- A rule (or production) that defines the start symbol/token and the possible end symbols/tokens

The symbol/token can be anything that is specific to the language that we consider.

For example:

- In the case of the English language, *a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z* are symbols/tokens/alphabets.
- In the case of the decimal numbering system *0, 1, 2, 3, 4, 5, 6, 7, 8, 9* are symbols/tokens/alphabets.

Generally, rules (or productions) are written in **Backus-Naur form (BNF)** notation.

Getting ready

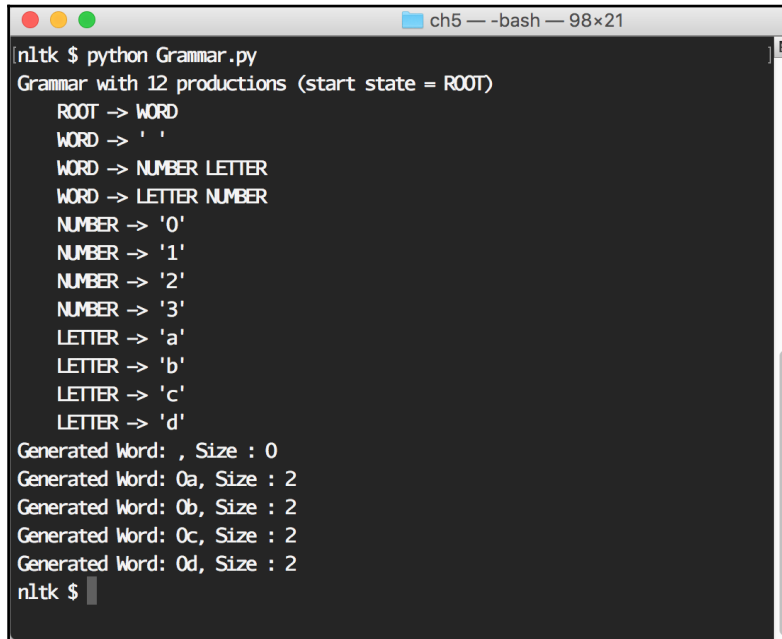
You should have a working Python (Python 3.6 is preferred) installed on your system, along with the NLTK library.

How to do it...

1. Open your atom editor (or your favorite programming editor).
2. Create a new file called `Grammar.py`.
3. Type the following source code:

```
Grammar.py
1 import nltk
2 import string
3 from nltk.parse.generate import generate
4 import sys
5
6 productions = [
7     "ROOT -> WORD",
8     "WORD -> ' '",
9     "WORD -> NUMBER LETTER",
10    "WORD -> LETTER NUMBER",
11 ]
12
13 digits = list(string.digits)
14 for digit in digits[:4]:
15     productions.append("NUMBER -> '{w}'".format(w=digit))
16
17 letters = " | ".join(list(string.ascii_lowercase)[:4])
18 productions.append("LETTER -> '{w}'".format(w=letters))
19
20 grammarString = "\n".join(productions)
21
22 grammar = nltk.CFG.fromstring(grammarString)
23
24 print(grammar)
25
26 for sentence in generate(grammar, n=5, depth=5):
27     palindrome = "".join(sentence).replace(" ", "")
28     print("Generated Word: {}, Size : {}".format(palindrome, len(palindrome)))
29
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```
ch5 — -bash — 98x21
[nltk $ python Grammar.py
Grammar with 12 productions (start state = ROOT)
ROOT -> WORD
WORD -> ' '
WORD -> NUMBER LETTER
WORD -> LETTER NUMBER
NUMBER -> '0'
NUMBER -> '1'
NUMBER -> '2'
NUMBER -> '3'
LETTER -> 'a'
LETTER -> 'b'
LETTER -> 'c'
LETTER -> 'd'
Generated Word: , Size : 0
Generated Word: 0a, Size : 2
Generated Word: 0b, Size : 2
Generated Word: 0c, Size : 2
Generated Word: 0d, Size : 2
nltk $
```

How it works...

Now, let's go through the program that we have just written and dig into the details:

```
import nltk
```

We are importing the `nltk` library into the current program:

```
import string
```

This instruction imports the `string` module into the current program:

```
from nltk.parse.generate import generate
```

This instruction imports the `generate` function from the `nltk.parse.generate` module, which helps in generating strings from the CFG that we are going to create:

```
productions = [  
    "ROOT -> WORD",  
    "WORD -> ' '",  
    "WORD -> NUMBER LETTER",  
    "WORD -> LETTER NUMBER",  
]
```

We are defining a new grammar here. The grammar can contain the following production rules:

- The starting symbol is `ROOT`
- The `ROOT` symbol can produce `WORD` symbol
- The `WORD` symbol can produce `' '` (empty space); this is a dead end production rule
- The `WORD` symbol can produce `NUMBER` symbol followed by `LETTER` symbol
- The `WORD` symbol can produce `LETTER` symbol followed by `NUMBER` symbol

These instructions further extend the production rules.

```
digits = list(string.digits)  
for digit in digits[:4]:  
    productions.append("NUMBER -> '{w}'".format(w=digit))
```

- `NUMBER` can produce terminal alphabets 0, 1, 2, or 3:

These instructions further extend the production rules.

```
letters = "" | "".join(list(string.ascii_lowercase)[:4])  
productions.append("LETTER -> '{w}'".format(w=letters))
```

- `LETTER` can produce lowercase alphabets a, b, c, or d.

Let's try to understand what this grammar is for. This grammar represents the language wherein there are words such as 0a, 1a, 2a, a1, a3, and so on.

All the production rules that we have stored so far in the list variable called `productions` are converted to a string:

```
grammarString = "\n".join(productions)
```

We are creating a new grammar object using the `nltk.CFG.fromstring()` method, which takes the `grammarString` variable that we have just created:

```
grammar = nltk.CFG.fromstring(grammarString)
```

These instructions print the first five auto generated words that are present in this language, which is defined with the grammar:

```
for sentence in generate(grammar, n=5, depth=5):
    palindrome = "".join(sentence).replace(" ", "")
    print("Generated Word: {}, Size : {}".format(palindrome,
        len(palindrome)))
```

Writing a probabilistic CFG

Probabilistic CFG is a special type of CFG in which the sum of all the probabilities for the non-terminal tokens (left-hand side) should be equal to one.

Let's write a simple example to understand more.

Getting ready

You should have a working Python (Python 3.6 is preferred) installed on your system, along with the NLTK library.

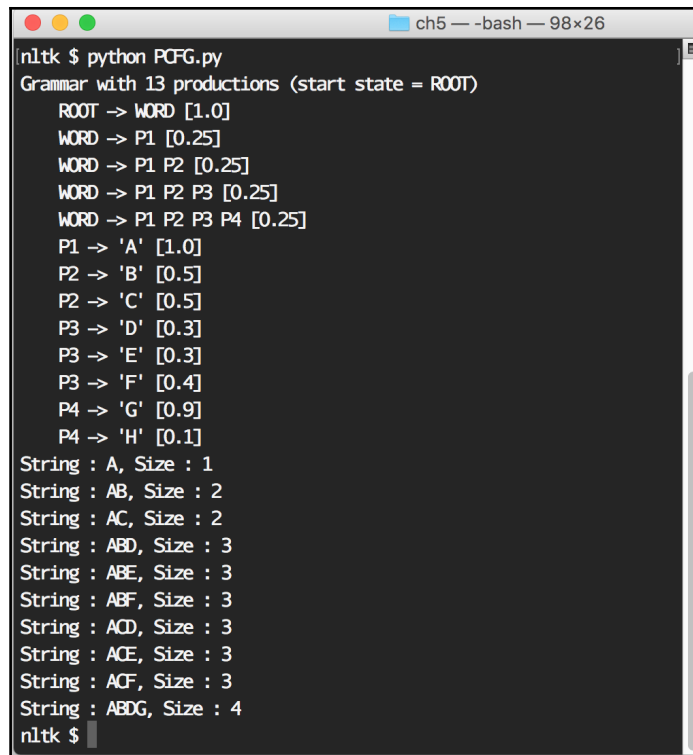
How to do it...

1. Open your atom editor (or your favorite programming editor).
2. Create a new file called `PCFG.py`.
3. Type the following source code:

```
PCFG.py
1  import nltk
2  from nltk.parse.generate import generate
3
4  productions = [
5      "ROOT -> WORD [1.0]",
6      "WORD -> P1 [0.25]",
7      "WORD -> P1 P2 [0.25]",
8      "WORD -> P1 P2 P3 [0.25]",
9      "WORD -> P1 P2 P3 P4 [0.25]",
10     "P1 -> 'A' [1.0]",
11     "P2 -> 'B' [0.5]",
12     "P2 -> 'C' [0.5]",
13     "P3 -> 'D' [0.3]",
14     "P3 -> 'E' [0.3]",
15     "P3 -> 'F' [0.4]",
16     "P4 -> 'G' [0.9]",
17     "P4 -> 'H' [0.1]",
18 ]
19
20 grammarString = "\n".join(productions)
21
22 grammar = nltk.PCFG.fromstring(grammarString)
23
24 print(grammar)
25
26 for sentence in generate(grammar, n=10, depth=5):
27     palindrome = "".join(sentence).replace(" ", "")
28     print("String : {}, Size : {}".format(palindrome, len(palindrome)))
29
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



```
ch5 — -bash — 98x26
nlTK $ python PCFG.py
Grammar with 13 productions (start state = ROOT)
  ROOT -> WORD [1.0]
  WORD -> P1 [0.25]
  WORD -> P1 P2 [0.25]
  WORD -> P1 P2 P3 [0.25]
  WORD -> P1 P2 P3 P4 [0.25]
  P1 -> 'A' [1.0]
  P2 -> 'B' [0.5]
  P2 -> 'C' [0.5]
  P3 -> 'D' [0.3]
  P3 -> 'E' [0.3]
  P3 -> 'F' [0.4]
  P4 -> 'G' [0.9]
  P4 -> 'H' [0.1]
String : A, Size : 1
String : AB, Size : 2
String : AC, Size : 2
String : AED, Size : 3
String : ABE, Size : 3
String : ABF, Size : 3
String : ACD, Size : 3
String : ACE, Size : 3
String : ACF, Size : 3
String : ABDG, Size : 4
nlTK $
```

How it works...

Now, let's go through the program that we have just written and dig into the details:

```
import nltk
```

This instruction imports the `nltk` module into our program:

```
from nltk.parse.generate import generate
```


This instruction imports the `generate` function from the `nltk.parse.generate` module:

```
productions = [
    "ROOT -> WORD [1.0]",
    "WORD -> P1 [0.25]",
    "WORD -> P1 P2 [0.25]",
    "WORD -> P1 P2 P3 [0.25]",
    "WORD -> P1 P2 P3 P4 [0.25]",
    "P1 -> 'A' [1.0]",
    "P2 -> 'B' [0.5]",
    "P2 -> 'C' [0.5]",
    "P3 -> 'D' [0.3]",
    "P3 -> 'E' [0.3]",
    "P3 -> 'F' [0.4]",
    "P4 -> 'G' [0.9]",
    "P4 -> 'H' [0.1]",
]
```

Here, we are defining the grammar for our language, which goes like this:

Description	Content
Starting symbol	ROOT
Non-terminals	WORD, P1, P2, P3, P4
Terminals	'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'

Once we have identified the tokens in the grammar, let's see what the production rules look like:

- There is a `ROOT` symbol, which is the starting symbol for this grammar
- There is a `WORD` symbol that has a probability of `1.0`
- There is a `WORD` symbol that can produce `P1` with a probability of `0.25`
- There is a `WORD` symbol that can produce `P1 P2` with a probability of `0.25`
- There is a `WORD` symbol that can produce `P1 P2 P3` with a probability of `0.25`
- There is a `WORD` symbol that can produce `P1 P2 P3 P4` with a probability of `0.25`
- The `P1` symbol can produce symbol `'A'` with a `1.0` probability
- The `P2` symbol can produce symbol `'B'` with a `0.5` probability
- The `P2` symbol can produce symbol `'C'` with a `0.5` probability
- The `P3` symbol can produce symbol `'D'` with a `0.3` probability

- The P_3 symbol can produce symbol 'E' with a 0.3 probability
- The P_3 symbol can produce symbol 'F' with a 0.4 probability
- The P_4 symbol can produce symbol 'G' with a 0.9 probability
- The P_4 symbol can produce symbol 'H' with a 0.1 probability

If you observe carefully, the sum of all the probabilities of the non-terminal symbols is equal to 1.0. This is a mandatory requirement for the PCFG.

We are joining the list of all the production rules into a string called the `grammarString` variable:

```
grammarString = "\n".join(productions)
```

This instruction creates a `grammar` object using the `nltk.PCFG.fromstring` method and taking the `grammarString` as input:

```
grammar = nltk.PCFG.fromstring(grammarString)
```

This instruction uses the Python built-in `print()` function to display the contents of the `grammar` object on screen. This will summarize the total number of tokens and production rules we have in the grammar that we have just created:

```
print(grammar)
```

We are printing 10 strings from this grammar using the NLTK built-in function `generate` and then displaying them on screen:

```
for sentence in generate(grammar, n=10, depth=5):  
    palindrome = "".join(sentence).replace(" ", "")  
    print("String : {}, Size : {}".format(palindrome, len(palindrome)))
```

Writing a recursive CFG

Recursive CFGs are a special types of CFG where the Tokens on the left-hand side are present on the right-hand side of a production rule.

Palindromes are the best examples of recursive CFG. We can always write a recursive CFG for palindromes in a given language.

To understand more, let's consider a language system with alphabets 0 and 1; so palindromes can be expressed as follows:

- 11
- 1001
- 010010

No matter in whatever direction we read these alphabets (left to right or right to left), we always get the same value. This is the special feature of palindromes.

In this recipe, we will write grammar to represent these palindromes and generate a few palindromes using the NLTK built-in string generation libraries.

Let's write a simple example to understand more.

Getting ready

You should have a working Python (Python 3.6 is preferred) installed on your system, along with the NLTK library.

How to do it...

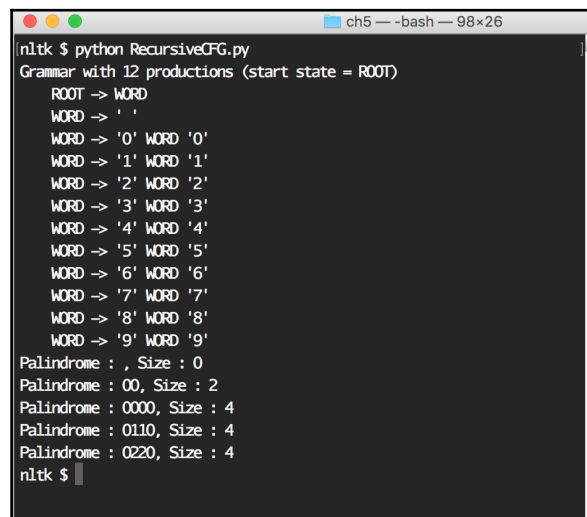
1. Open your atom editor (or your favorite programming editor).
2. Create a new file called `RecursiveCFG.py`.
3. Type the following source code:

```

RecursiveCFG.py
1  import nltk
2  import string
3  from nltk.parse.generate import generate
4
5  productions = [
6      "ROOT -> WORD",
7      "WORD -> ' '"
8  ]
9
10 alphabets = list(string.digits)
11
12 for alphabet in alphabets:
13     productions.append("WORD -> '{w}' WORD '{w}'".format(w=alphabet))
14
15 grammarString = "\n".join(productions)
16
17 grammar = nltk.CFG.fromstring(grammarString)
18
19 print(grammar)
20
21 for sentence in generate(grammar, n=5, depth=5):
22     palindrome = "".join(sentence).replace(" ", "")
23     print("Palindrome : {}, Size : {}".format(palindrome, len(palindrome)))
24

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```

nltk $ python RecursiveCFG.py
Grammar with 12 productions (start state = ROOT)
  ROOT -> WORD
  WORD -> ' '
  WORD -> '0' WORD '0'
  WORD -> '1' WORD '1'
  WORD -> '2' WORD '2'
  WORD -> '3' WORD '3'
  WORD -> '4' WORD '4'
  WORD -> '5' WORD '5'
  WORD -> '6' WORD '6'
  WORD -> '7' WORD '7'
  WORD -> '8' WORD '8'
  WORD -> '9' WORD '9'
Palindrome : , Size : 0
Palindrome : 00, Size : 2
Palindrome : 0000, Size : 4
Palindrome : 0110, Size : 4
Palindrome : 0220, Size : 4
nltk $

```

How it works...

Now, let's go through the program that we have just written and dig into the details. We are importing the `nltk` library into our program for future use:

```
import nltk
```

We are also importing the `string` library into our program for future use:

```
import string
```

We are importing the `generate` function from the `nltk.parse.generate` module:

```
from nltk.parse.generate import generate
```

We have created a new list data structure called `productions`, where there are two elements. Both the elements are strings that represent the two productions in our CFG:

```
productions = [  
    "ROOT -> WORD",  
    "WORD -> ' '"  
]
```

We are retrieving the list of decimal digits as a list in the `alphabets` variable:

```
alphabets = list(string.digits)
```

Using the digits 0 to 9, we add more productions to our list. These are the production rules that define palindromes:

```
for alphabet in alphabets:  
    productions.append("WORD -> '{w}' WORD '{w}'".format(w=alphabet))
```

Once all the rules are generated, we concatenate them as strings to a variable, `grammarString`:

```
grammarString = "\n".join(productions)
```

In this instruction, we are creating a new `grammar` object by passing the newly constructed `grammarString` to the NLTK built-in `nltk.CFG.fromstring` function:

```
grammar = nltk.CFG.fromstring(grammarString)
```

In this instruction, we print the grammar that we have just created by calling the Python built-in `print()` function:

```
print(grammar)
```

We are generating five palindromes using the `generate` function of the NLTK library and printing the same on the screen:

```
for sentence in generate(grammar, n=5, depth=5):  
    palindrome = "".join(sentence).replace(" ", "")  
    print("Palindrome : {}, Size : {}".format(palindrome, len(palindrome)))
```

6

Chunking, Sentence Parse, and Dependencies

In this chapter, we will perform the following recipes:

- Using the built-in chunker
- Writing your own simple chunker
- Training a chunker
- Parsing recursive descent
- Parsing shift reduce
- Parsing dependency grammar and projective dependency
- Parsing a chart

Introduction

We have learned so far that the Python NLTK can be used to do **part-of-speech (POS)** recognition in a given piece of text. But sometimes we are interested in finding more details about the text that we are dealing with. For example, I might be interested in finding the names of some famous personalities, places, and so on in a given text. We can maintain a very big dictionary of all these names. But in the simplest form, we can use a POS analysis to identify these patterns very easily.

Chunking is the process of extracting short phrases from text. We will leverage POS tagging algorithms to do chunking. Remember that the tokens (words) produced by chunking do not overlap.

Using the built-in chunker

In this recipe, we will learn how to use the in-built chunker. These are the features that will be used from NLTK as part of this process:

- Punkt tokenizer (default)
- Averaged perception tagger (default)
- Maxent NE chunker (default)

Getting ready

You should have Python installed along with the `nltk` library. Prior understanding of POS tagging as explained in Chapter 5, *POS Tagging and Grammars* is good to have.

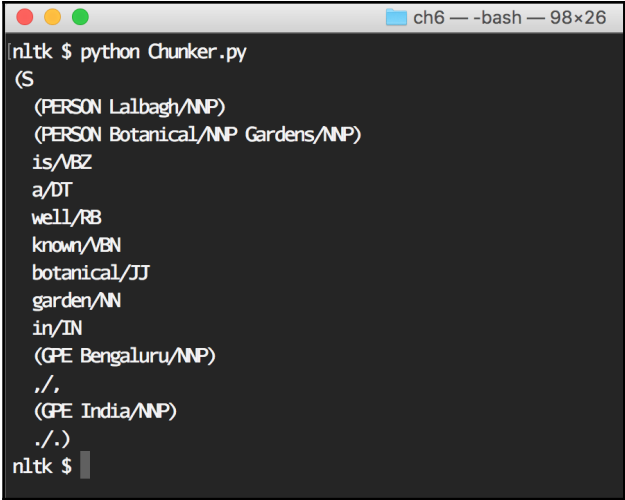
How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `Chunker.py`.
3. Type the following source code:

```
Chunker.py
1 import nltk
2
3 text = "Lalbagh Botanical Gardens is a well known botanical garden in Bengaluru, India."
4 sentences = nltk.sent_tokenize(text)
5 for sentence in sentences:
6     words = nltk.word_tokenize(sentence)
7     tags = nltk.pos_tag(words)
8     chunks = nltk.ne_chunk(tags)
9     print(chunks)
10
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



```
ch6 — -bash — 98x26
nltk $ python Chunker.py
(S
 (PERSON Lalbagh/NNP)
 (PERSON Botanical/NNP Gardens/NNP)
 is/VBZ
 a/DT
 well/RB
 known/VBN
 botanical/JJ
 garden/NN
 in/IN
 (GPE Bengaluru/NNP)
 ./,
 (GPE India/NNP)
 ./.)
nltk $
```

How it works...

Let's try to understand how the program works. This instruction imports the `nltk` module into the program:

```
import nltk
```

This is the data that we are going to analyze as part of this recipe. We are adding this string to a variable called `text`:

```
text = "Lalbagh Botanical Gardens is a well known botanical garden in  
Bengaluru, India."
```

This instruction is going to break the given text into multiple sentences. The result is a list of sentences stored in the `sentences` variable:

```
sentences = nltk.sent_tokenize(text)
```

In this instruction, we are looping through all the sentences that we have extracted. Each sentence is stored in the `sentence` variable:

```
for sentence in sentences:
```

This instruction breaks the sentence into non-overlapping words. The result is stored in a variable called `words`:

```
words = nltk.word_tokenize(sentence)
```

In this instruction, we do POS analysis using the default tagger that is available with NLTK. Once the identification is done, the result is stored in a variable called `tags`:

```
tags = nltk.pos_tag(words)
```

In this instruction, we call the `nltk.ne_chunk()` function, which does the chunking part for us. The result is stored in a variable called `chunks`. The result is actually tree-structured data that contains the paths of the tree:

```
chunks = nltk.ne_chunk(tags)
```

This prints the chunks that are identified in the given input string. Chunks are grouped in brackets, '(' and ')', to easily distinguish them from other words that are in the input text.

```
print(chunks)
```

Writing your own simple chunker

In this recipe, we will write our own Regex chunker. Since we are going to use regular expressions to write this chunker, we need to understand a few differences in the way we write regular expressions for chunking.

In [Chapter 4, *Regular Expressions*](#), we understood regular expressions and how to write them. For example, a regular expression of the form `[a-z, A-Z]+` matches all words in a sentence that is written in English.

We already understand that by using NLTK, we can identify the POS in their short form (tags such as `V`, `NN`, `NNP`, and so on). Can we write regular expressions using these POS?

The answer is yes. You have guessed it correctly. We can leverage POS-based regular expression writing. Since we are using POS tags to write these regular expressions, they are called tag patterns.

Just like the way we write the native alphabets (a-z) of a given natural language to match various patterns, we can also leverage POS to match words (any combinations from dictionary) according to the NLTK matched POS.

These tag patterns are one of the most powerful features of NLTK because they give us the flexibility to match the words in a sentence just by POS-based regular expressions.

In order to learn more about these, let's dig further:

```
"Ravi is the CEO of a Company. He is very powerful public speaker also."
```

Once we identify the POS, this is how the result looks:

```
[('Ravi', 'NNP'), ('is', 'VBZ'), ('the', 'DT'), ('CEO', 'NNP'), ('of',  
'IN'), ('a', 'DT'), ('Company', 'NNP'), ('.', '.')] [  
(('He', 'PRP'), ('is', 'VBZ'), ('very', 'RB'), ('powerful', 'JJ'),  
(('public', 'JJ'), ('speaker', 'NN'), ('also', 'RB'), ('.', '.')]
```

Later, we can use this information to extract the noun phrases.

Let's pay close attention to the preceding POS output. We can make the following observations:

- Chunks are one or more continuous NNP
- Chunks are NNP followed by a DT
- Chunks are NP followed by one more JJ

By using these three simple observations, let's write a regular expression using POS, which is called as tag phrase in the BNF form:

```
NP -> <PRP>  
NP -> <DT>*<NNP>  
NP -> <JJ>*<NN>  
NP -> <NNP>+
```

We are interested in extracting the following chunks from the input text:

- Ravi
- the CEO
- a company
- powerful public speaker

Let's write a simple Python program that gets the job done.

Getting ready

You should have Python installed, along with the `nltk` library. A fair understanding of regular expressions is good to have.

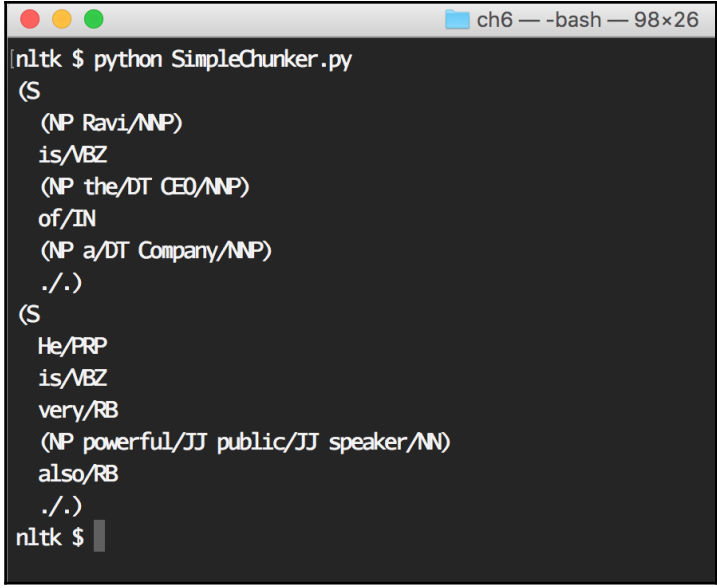
How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `SimpleChunker.py`.
3. Type the following source code:

```
SimpleChunker.py
1 import nltk
2
3 text = "Ravi is the CEO of a Company. He is very powerful public speaker also."
4
5 grammar = '\n'.join([
6     'NP: {<DT>*<NNP>}',
7     'NP: {<JJ>*<NN>}',
8     'NP: {<NNP>+}',
9 ])
10
11 sentences = nltk.sent_tokenize(text)
12
13 for sentence in sentences:
14     words = nltk.word_tokenize(sentence)
15     tags = nltk.pos_tag(words)
16     chunkparser = nltk.RegexpParser(grammar)
17     result = chunkparser.parse(tags)
18     print(result)
19
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



```
ch6 — -bash — 98x26
[nltk $ python SimpleChunker.py
(S
  (NP Ravi/NNP)
  is/VBZ
  (NP the/DT CEO/NNP)
  of/IN
  (NP a/DT Company/NNP)
  ./.)
(S
  He/PRP
  is/VBZ
  very/RB
  (NP powerful/JJ public/JJ speaker/NN)
  also/RB
  ./.)
nltk $
```

How it works...

Now, let's understand how the program works:

This instruction imports the `nltk` library into the current program:

```
import nltk
```

We are declaring the `text` variable with the sentences that we want to process:

```
text = "Ravi is the CEO of a Company. He is very powerful public speaker  
also."
```

In this instruction, we are writing regular expressions, which are written using POS; so they are specially called tag patterns. These tag patterns are not a randomly created ones. They are carefully crafted from the preceding example.

```
grammar = '\n'.join([
    'NP: {<DT>*<NNP>}',
    'NP: {<JJ>*<NN>}',
    'NP: {<NNP>+}',
])
```

Let's understand these tag patterns:

- NP is followed by one or more <DT> and then an <NNP>
- NP is followed by one or more <JJ> and then an <NN>
- NP is one more <NNP>

The more text we process, the more rules like this we can discover. These are specific to the language we process. So, this is a practice we should do in order to become more powerful at information extraction:

```
sentences = nltk.sent_tokenize(text)
```

First we break the input text into sentences by using the `nltk.sent_tokenize()` function:

```
for sentence in sentences:
```

This instruction iterates through a list of all sentences and assigns one sentence to the `sentence` variable:

```
words = nltk.word_tokenize(sentence)
```

This instruction breaks the sentence into tokens using the `nltk.word_tokenize()` function and puts the result into the `words` variable:

```
tags = nltk.pos_tag(words)
```

This instruction does the POS identification on the `words` variable (which has a list of words) and puts the result in the `tags` variable (which has each word correctly tagged with its respective POS tag):

```
chunkparser = nltk.RegexpParser(grammar)
```

This instruction invokes the `nltk.RegexpParser` on the grammar that we have created before. The object is available in the `chunkparser` variable:

```
result = chunkparser.parse(tags)
```

We parse the tags using the object and the result is stored in the `result` variable:

```
print(result)
```

Now, we display the identified chunks on screen using the `print()` function. The output is a tree structure with words and their associated POS.

Training a chunker

In this recipe, will learn the training process, training our own chunker, and evaluating it. Before we go into training, we need to understand the type of data we are dealing with. Once we have a fair understanding of the data, we must train it according to the pieces of information we need to extract. One particular way of training the data is to use IOB tagging for the chunks that we extract from the given text.

Naturally, we find different words in a sentence. From these words, we can find POS. Later, when chunking the text, we need to further tag the words according to where they are present in the text.

Take the following example:

"Bill Gates announces Satya Nadella as new CEO of Microsoft"

Once we've done POS tagging and hunking of the data, we will see an output similar to this one:

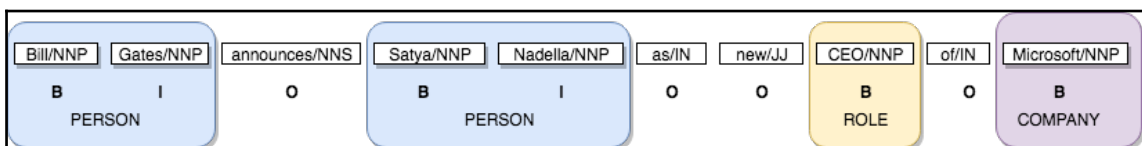
```

Bill NNP B-PERSON
Gates NNP I-PERSON
announces NNS O
Satya NNP B-PERSON
Nadella NNP I-PERSON
as IN O
new JJ O
CEO NNP B-ROLE
of IN O
Microsoft NNP B-COMPANY
    
```

This is called the IOB format, where each line consists of three tokens separated by spaces.

Column	Description
First column in IOB	The actual word in the input sentence
Second column in IOB	The POS for the word
Third column in IOB	Chunk identifier with I (inside chunk), O (outside chunk), B (beginning word of the chunk), and the appropriate suffix to indicate the category of the word

Let's see this in a diagram:



Once we have the training data in IOB format, we can further use it to extend the reach of our chunker by applying it to other datasets. Training is very expensive if we want to do it from scratch or want to identify new types of keywords from the text.

Let's try to write a simple chunker using the `regexparser` and see what types of results it gives.

Getting ready

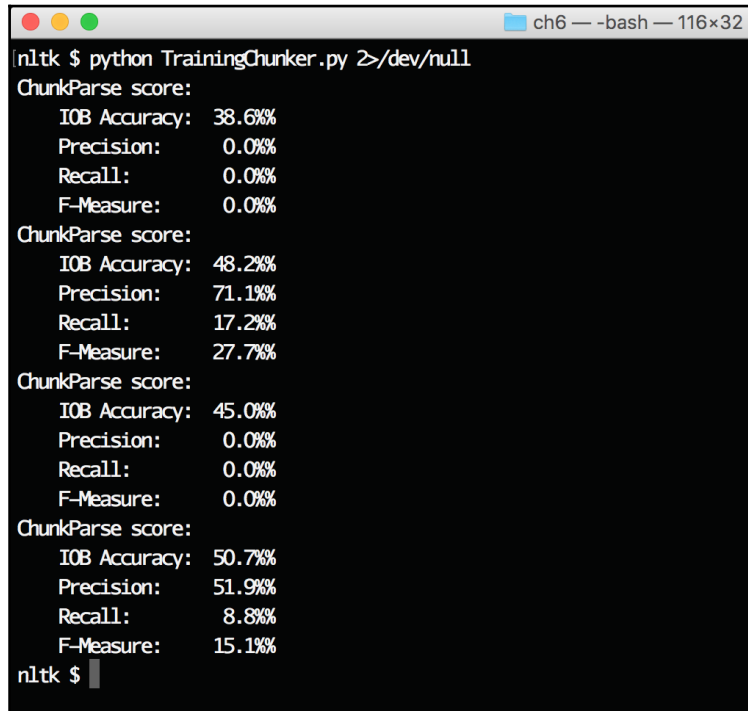
You should have Python installed, along with the `nltk` library.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `TrainingChunker.py`.
3. Type the following source code:

```
TrainingChunker.py
1 import nltk
2 from nltk.corpus import conll2000
3 from nltk.corpus import treebank_chunk
4
5 def mySimpleChunker():
6     grammar = 'NP: {<NNP>+}'
7     return nltk.RegexpParser(grammar)
8
9 def test_nothing(data):
10     cp = nltk.RegexpParser("")
11     print(cp.evaluate(data))
12
13 def test_mysimplechunker(data):
14     schunker = mySimpleChunker()
15     print(schunker.evaluate(data))
16
17
18 datasets = [
19     conll2000.chunked_sents('test.txt', chunk_types=['NP']),
20     treebank_chunk.chunked_sents()
21 ]
22
23 for dataset in datasets:
24     test_nothing(dataset[:50])
25     test_mysimplechunker(dataset[:50])
26
```


4. Save the file.
5. Run the program using the Python interpreter.
6. You will see this output:

A terminal window titled 'ch6 — -bash — 116x32' displays the output of a Python script. The script runs 'python TrainingChunker.py >/dev/null' in an 'nltk \$' prompt. The output shows four sets of 'ChunkParse score' metrics. Each set includes 'IOB Accuracy', 'Precision', 'Recall', and 'F-Measure' percentages. The scores for the four sets are: (38.6%, 0.0%, 0.0%, 0.0%), (48.2%, 71.1%, 17.2%, 27.7%), (45.0%, 0.0%, 0.0%, 0.0%), and (50.7%, 51.9%, 8.8%, 15.1%). The terminal ends with an 'nltk \$' prompt.

```
nltk $ python TrainingChunker.py >/dev/null
ChunkParse score:
  IOB Accuracy: 38.6%
  Precision:    0.0%
  Recall:       0.0%
  F-Measure:    0.0%
ChunkParse score:
  IOB Accuracy: 48.2%
  Precision:    71.1%
  Recall:       17.2%
  F-Measure:    27.7%
ChunkParse score:
  IOB Accuracy: 45.0%
  Precision:    0.0%
  Recall:       0.0%
  F-Measure:    0.0%
ChunkParse score:
  IOB Accuracy: 50.7%
  Precision:    51.9%
  Recall:       8.8%
  F-Measure:    15.1%
nltk $
```

How it works...

This instruction imports the `nltk` module into the current program:

```
import nltk
```

This instruction imports the `conll2000` corpus into the current program:

```
from nltk.corpus import conll2000
```

This instruction imports the `treebank` corpus into the current program:

```
from nltk.corpus import treebank_chunk
```

We are defining a new function, `mySimpleChunker()`. We are also defining a simple tag pattern that extracts all the words that have POS of `NNP` (proper nouns). This grammar is used for our chunker to extract the named entities:

```
def mySimpleChunker():
    grammar = 'NP: {<NNP>+}'
    return nltk.RegexpParser(grammar)
```

This is a simple chunker; it doesn't extract anything from the given text. Useful to see if the algorithm works correctly:

```
def test_nothing(data):
    cp = nltk.RegexpParser("")
    print(cp.evaluate(data))
```

This function uses `mySimpleChunker()` on the test data and evaluates the accuracy of the data with respect to already tagged input data:

```
def test_mysimplechunker(data):
    schunker = mySimpleChunker()
    print(schunker.evaluate(data))
```

We create a list of two datasets, one from `conll2000` and another from `treebank`:

```
datasets = [
    conll2000.chunked_sents('test.txt', chunk_types=['NP']),
    treebank_chunk.chunked_sents()
]
```

We iterate over the two datasets and call `test_nothing()` and `test_mysimplechunker()` on the first 50-IOB tagged sentences to see what the accuracy of the chunker looks like.

```
for dataset in datasets:
    test_nothing(dataset[:50])
    test_mysimplechunker(dataset[:50])
```

Parsing recursive descent

Recursive descent parsers belong to the family of parsers that read the input from left to right and build the parse tree in a top-down fashion and traversing nodes in a pre-order fashion. Since the grammar itself is expressed using CFG methodology, the parsing is recursive in nature. This kind of parsing technique is used to build compilers for parsing instructions of programming languages.

In this recipe, we will explore how we can use the RD parser that comes with the NLTK library.

Getting ready

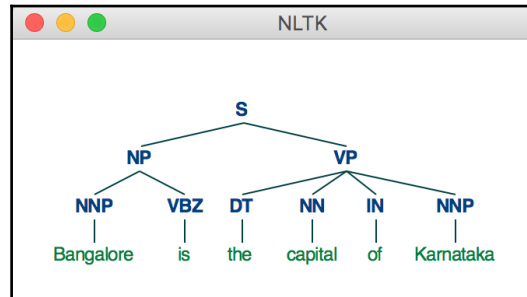
You should have Python installed, along with the `nltk` library.

How to do it...

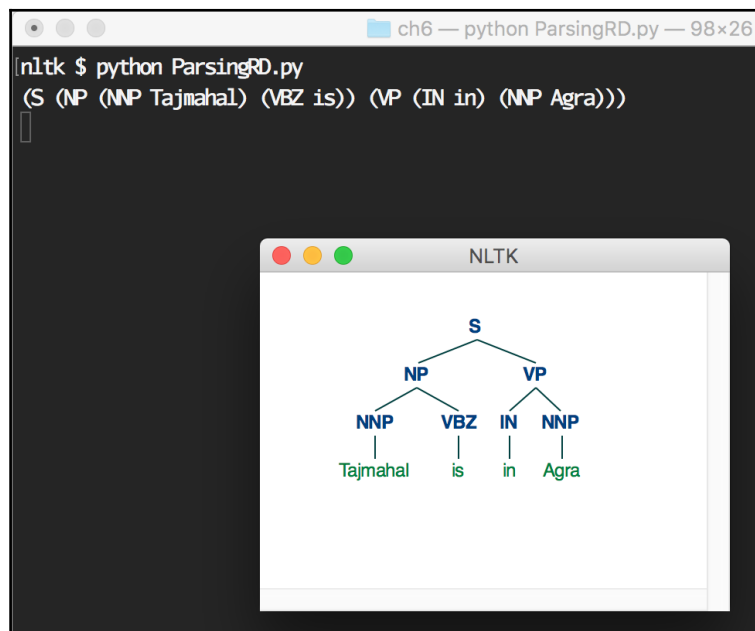
1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `ParsingRD.py`.
3. Type the following source code:

```
ParsingRD.py
1 import nltk
2
3 def RDParseExample(grammar, textlist):
4     parser = nltk.parse.RecursiveDescentParser(grammar)
5     for text in textlist:
6         sentence = nltk.word_tokenize(text)
7         for tree in parser.parse(sentence):
8             print(tree)
9             tree.draw()
10
11 grammar = nltk.CFG.fromstring("""
12 S -> NP VP
13 NP -> NNP VBZ
14 VP -> IN NNP | DT NN IN NNP
15 NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
16 VBZ -> 'is'
17 IN -> 'in' | 'of'
18 DT -> 'the'
19 NN -> 'capital'
20 """)
21
22 text = [
23     "Tajmahal is in Agra",
24     "Bangalore is the capital of Karnataka",
25 ]
26
27 RDParseExample(grammar, text)
28
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



This graph is the output of the second sentence in the input as parsed by the RD parser:



How it works...

Let's see how the program works. In this instruction, we are importing the `nltk` library:

```
import nltk
```

In these instructions, we are defining a new function, `SRParserExample`; it takes a grammar object and `textlist` as parameters:

```
def RDParserExample(grammar, textlist):
```

We are creating a new parser object by calling `RecursiveDescentParser` from the `nltk.parse` library. We pass `grammar` to this class for initialization:

```
parser = nltk.parse.RecursiveDescentParser(grammar)
```

In these instructions, we are iterating over the list of sentences in the `textlist` variable. Each text item is tokenized using the `nltk.word_tokenize()` function and then the resultant words are passed to the `parser.parse()` function. Once the parse is complete, we display the result on the screen and also show the parse tree:

```
for text in textlist:
    sentence = nltk.word_tokenize(text)
    for tree in parser.parse(sentence):
        print(tree)
        tree.draw()
```

We create a new CFG object using `grammar`:

```
grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> NNP VBZ
VP -> IN NNP | DT NN IN NNP
NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
VBZ -> 'is'
IN -> 'in' | 'of'
DT -> 'the'
NN -> 'capital'
""")
```

These are the two sample sentences we use to understand the parser:

```
text = [
    "Tajmahal is in Agra",
    "Bangalore is the capital of Karnataka",
]
```

We call `RDParserExample` using the `grammar` object and the list of sample sentences.

```
RDParserExample(grammar, text)
```

Parsing shift-reduce

In this recipe, we will learn to use and understand shift-reduce parsing.

Shift-reduce parsers are special types of parsers that parse the input text from left to right on a single line sentences and top to bottom on multiline sentences.

For every alphabet/token in the input text, this is how parsing happens:

- Read the first token from the input text and push it to the stack (shift operation)
- Read the complete parse tree on the stack and see which production rule can be applied, by reading the production rule from right to left (reduce operation)
- This process is repeated until we run out of production rules, when we accept that parsing has failed
- This process is repeated until all of the input is consumed; we say parsing has succeeded

In the following examples, we see that only one input text is going to be parsed successfully and the other cannot be parsed.

Getting ready

You should have Python installed, along with the `nltk` library. An understanding of writing grammars is needed.

How to do it...

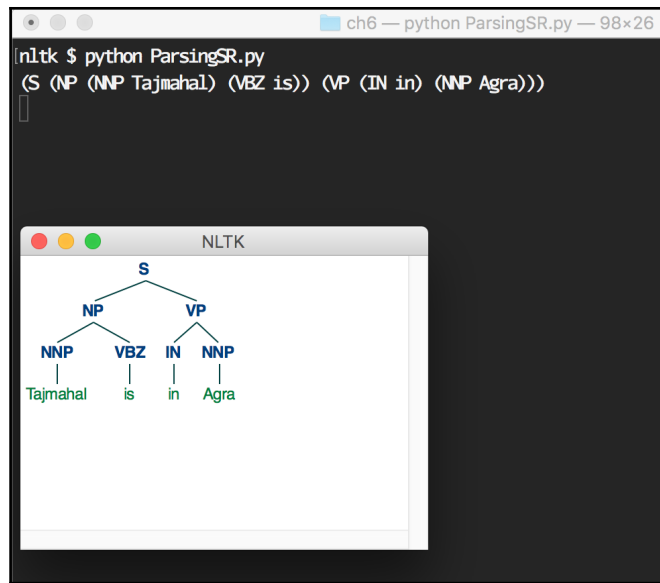
1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `ParsingSR.py`.

3. Type the following source code:

```
ParsingSR.py
1 import nltk
2
3 def SRParserExample(grammar, textlist):
4     parser = nltk.parse.ShiftReduceParser(grammar)
5     for text in textlist:
6         sentence = nltk.word_tokenize(text)
7         for tree in parser.parse(sentence):
8             print(tree)
9             tree.draw()
10
11 text = [
12     "Tajmahal is in Agra",
13     "Bangalore is the capital of Karnataka",
14 ]
15
16 grammar = nltk.CFG.fromstring("""
17 S -> NP VP
18 NP -> NNP VBZ
19 VP -> IN NNP | DT NN IN NNP
20 NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
21 VBZ -> 'is'
22 IN -> 'in' | 'of'
23 DT -> 'the'
24 NN -> 'capital'
25 """)
26
27 SRParserExample(grammar, text)
28
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



How it works...

Let's see how the program works. In this instruction we are importing the `nltk` library:

```
import nltk
```

In these instructions, we are defining a new function, `SRParserExample`; it takes a grammar object and `textlist` as parameters:

```
def SRParserExample(grammar, textlist):
```

We are creating a new parser object by calling `ShiftReduceParser` from the `nltk.parse` library. We pass `grammar` to this class for initialization:

```
parser = nltk.parse.ShiftReduceParser(grammar)
```


In these instructions, we are iterating over the list of sentences in the `textlist` variable. Each text item is tokenized using the `nltk.word_tokenize()` function and then the resultant words are passed to the `parser.parse()` function. Once the parse is complete, we display the result on the screen and also show the parse tree:

```
for text in textlist:
    sentence = nltk.word_tokenize(text)
    for tree in parser.parse(sentence):
        print(tree)
        tree.draw()
```

These are the two sample sentences we are using to understand the shift-reduce parser:

```
text = [
    "Tajmahal is in Agra",
    "Bangalore is the capital of Karnataka",
]
```

We create a new CFG object using the grammar:

```
grammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> NNP VBZ
VP -> IN NNP | DT NN IN NNP
NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
VBZ -> 'is'
IN -> 'in' | 'of'
DT -> 'the'
NN -> 'capital'
""")
```

We call the `SRParserExample` using the grammar object and the list of sample sentences.

```
SRParserExample(grammar, text)
```

Parsing dependency grammar and projective dependency

In this recipe, we will learn how to parse dependency grammar and use it with the projective dependency parser.

Dependency grammars are based on the concept that sometimes there are direct relationships between words that form a sentence. The example in this recipe shows this clearly.

Getting ready

You should have Python installed, along with the `nltk` library.

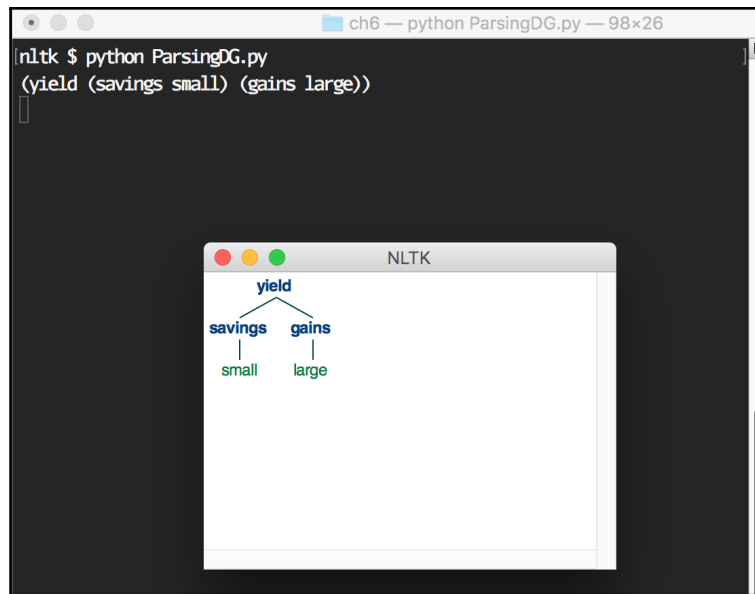
How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `ParsingDG.py`.
3. Type the following source code:

```
ParsingDG.py
1  import nltk
2
3  grammar = nltk.grammar.DependencyGrammar.fromstring("""
4  'savings' -> 'small'
5  'yield' -> 'savings'
6  'gains' -> 'large'
7  'yield' -> 'gains'
8  """)
9
10 sentence = 'small savings yield large gains'
11 dp = nltk.parse.ProjectiveDependencyParser(grammar)
12 for t in sorted(dp.parse(sentence.split())):
13     print(t)
14     t.draw()
15
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



How it works...

Let's see how the program works. This instruction imports the `nltk` library into the program:

```
import nltk
```

This instruction creates a grammar object using the `nltk.grammar.DependencyGrammar` class. We are adding the following productions to the grammar:

```
grammar = nltk.grammar.DependencyGrammar.fromstring("""
'savings' -> 'small'
'yield' -> 'savings'
'gains' -> 'large'
'yield' -> 'gains'
""")
```

Let's understand more about these productions:

- small related to savings
- savings related to yield
- large related to gains
- gains related to yield

This is the sample sentence on which we are going to run the parser. It is stored in a variable called `sentence`:

```
sentence = 'small savings yield large gains'
```

This instruction is creating a new `nltk.parse.ProjectiveDependencyParser` object using the grammar we have just defined:

```
dp = nltk.parse.ProjectiveDependencyParser(grammar)
```

We are doing many things in this for loop:

```
for t in sorted(dp.parse(sentence.split())):
    print(t)
    t.draw()
```

Preceding for loop does:

- We are breaking the words in the sentence
- All the list of words are fed to the `dp` object as input
- The result from the parsed output is sorted using the `sorted()` built-in function
- Iterate over all the tree paths and display them on screen as well as render the result in a beautiful tree form

Parsing a chart

Chart parsers are special types of parsers which are suitable for natural languages as they have ambiguous grammars. They use dynamic programming to generate the desired results.

The good thing about dynamic programming is that, it breaks the given problem into subproblems and stores the result in a shared location, which can be further used by algorithm wherever similar subproblem occurs elsewhere. This greatly reduces the need to re-compute the same thing over and over again.

In this recipe, we will learn the chart parsing features that are provided by the NLTK library.

Getting ready

You should have Python installed, along with the `nltk` library. An understanding of grammars is good to have.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `ParsingChart.py`.
3. Type the following source code:

```
ParsingChart.py
1 from nltk.grammar import CFG
2 from nltk.parse.chart import ChartParser, BU_LC_STRATEGY
3
4 grammar = CFG.fromstring("""
5 S -> T1 T4
6 T1 -> NNP VBZ
7 T2 -> DT NN
8 T3 -> IN NNP
9 T4 -> T3 | T2 T3
10 NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
11 VBZ -> 'is'
12 IN -> 'in' | 'of'
13 DT -> 'the'
14 NN -> 'capital'
15 """)
16
17 cp = ChartParser(grammar, BU_LC_STRATEGY, trace=True)
18
19 sentence = "Bangalore is the capital of Karnataka"
20 tokens = sentence.split()
21 chart = cp.chart_parse(tokens)
22 parses = list(chart.parses(grammar.start()))
23 print("Total Edges :", len(chart.edges()))
24 for tree in parses: print(tree)
25 tree.draw()
26
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:

```

nltk $ python ParsingChart.py
|.Bangal. is .the .capita. of .Karnat.|
| [ ] | . . . . . | [0:1] 'Bangalore'
| [ ] | . . . . . | [1:2] 'is'
| [ ] | . . . . . | [2:3] 'the'
| [ ] | . . . . . | [3:4] 'capital'
| [ ] | . . . . . | [4:5] 'of'
| [ ] | . . . . . | [5:6] 'Karnataka'
| [ ] | . . . . . | [0:1] NNP -> 'Bangalore' *
| [ ] | . . . . . | [0:1] T1 -> NNP * VBZ
| [ ] | . . . . . | [1:2] VBZ -> 'is' *
| [ ] | . . . . . | [0:2] T1 -> NNP VBZ *
| [ ] | . . . . . | [0:2] S -> T1 * T4
| [ ] | . . . . . | [2:3] DT -> 'the' *
| [ ] | . . . . . | [2:3] T2 -> DT * NN
| [ ] | . . . . . | [3:4] NN -> 'capital' *
| [ ] | . . . . . | [2:4] T2 -> DT NN *
| [ ] | . . . . . | [2:4] T4 -> T2 * T3
| [ ] | . . . . . | [4:5] IN -> 'of' *
| [ ] | . . . . . | [4:5] T3 -> IN * NNP
| [ ] | . . . . . | [5:6] NNP -> 'Karnataka' *
| [ ] | . . . . . | [5:6] T1 -> NNP * VBZ
| [ ] | . . . . . | [4:6] T3 -> IN NNP *
| [ ] | . . . . . | [4:6] T4 -> T3 *
| [ ] | . . . . . | [2:6] T4 -> T2 T3 *
| [ ] | . . . . . | [0:6] S -> T1 T4 *

Total Edges : 24
(S
  (T1 (NP (Bangalore)) (VBZ is))
  (T4 (T2 (DT the) (NN capital)) (T3 (IN of) (NP Karnataka))))

```

How it works...

Let's see how the program works. This instruction imports the CFG module into the program:

```
from nltk.grammar import CFG
```

This instruction imports the ChartParser and BU_LC_STRATEGY features into the program:

```
from nltk.parse.chart import ChartParser, BU_LC_STRATEGY
```

We are creating a sample grammar for the example that we are going to use. All the producers are expressed in the BNF form:

```
grammar = CFG.fromstring("""
S -> T1 T4
T1 -> NNP VBZ
T2 -> DT NN
T3 -> IN NNP
T4 -> T3 | T2 T3
NNP -> 'Tajmahal' | 'Agra' | 'Bangalore' | 'Karnataka'
VBZ -> 'is'
IN -> 'in' | 'of'
DT -> 'the'
NN -> 'capital'
""")
```

The grammar consists of:

- A starting token, *S*, which produces *T1 T4*
- Non-terminal tokens *T1*, *T2*, *T3*, and *T4*, which further produce *NNP VBZ*, *DT NN*, *IN NNP*, *T2*, or *T2 T3* respectively
- Terminal tokens, which are words from the English dictionary

A new chart parser object is created using the grammar object `BU_LC_STRATEGY`, and we have set `trace` to `True` so that we can see how the parsing happens on the screen:

```
cp = ChartParser(grammar, BU_LC_STRATEGY, trace=True)
```

We are going to process this sample string in this program; it is stored in a variable called `sentence`:

```
sentence = "Bangalore is the capital of Karnataka"
```

This instruction creates a list of words from the example sentence:

```
tokens = sentence.split()
```

This instruction takes the list of words as input and then starts the parsing. The result of the parsing is made available in the `chart` object:

```
chart = cp.chart_parse(tokens)
```

We are acquiring all the parse trees that are available in the chart into the `parses` variable:

```
parses = list(chart.parses(grammar.start()))
```

This instruction prints the total number of edges in the current `chart` object:

```
print("Total Edges :", len(chart.edges()))
```

This instruction prints all the parse trees on the screen:

```
for tree in parses: print(tree)
```

This instruction shows a nice tree view of the chart on a GUI widget.

```
tree.draw()
```


7

Information Extraction and Text Classification

In this chapter, we will cover the following recipes:

- Using inbuilt NERs
- Creating, inverting, and using dictionaries
- Creating your own NEs
- Choosing the feature set
- Segmenting sentences using classification
- Classifying documents
- Writing a POS tagger with context

Introduction

Information retrieval is a vast area and has many challenges. In previous chapters, we understood regular expressions, grammars, **Parts-of-Speech (POS)** tagging, and chunking. The natural step after this process is to identify the Interested Entities in a given piece of text. To be clear, when we are processing large amounts of data, we are really interested in finding out whether any famous personalities, places, products, and so on are mentioned. These things are called **named entities** in NLP. We will understand more about these with examples in the following recipes. Also, we will see how we can leverage the clues that are present in the input text to categorize large amounts of text, and many more examples will be explained. Stay tuned!

Understanding named entities

So far, we have seen how to parse the text, identify parts of speech, and extract chunks from the text. The next thing that we need to look into is finding **proper nouns**, which are also called named entities.

Named entities help us understand more about what is being referred to in a given text so that we can further classify the data. Since named entities comprise more than one word, it is sometimes difficult to find these from the text.

Let's take up the following examples to understand what named entities are:

Sentence	Named entities
Hampi is on the South Bank of Tungabhadra river	Hampi, Tungabhadra River
Paris is famous for Fashion	Paris
Burj Khalifa is one of the Skyscrapers in Dubai	Burj Khalifa , Dubai
Jeff Weiner is the CEO of LinkedIn	Jeff Weiner, LinkedIn

Let's take a closer look at these and try to understand:

1. Even though *South Bank* refers to a direction, it does not qualify as a named entity because we cannot uniquely identify the object from that.
2. Even though *Fashion* is a noun, we cannot completely qualify it as named entity.
3. *Skyscraper* is a noun, but there can be many possibilities for Skyscrapers.
4. *CEO* is a role here; there are many possible persons who can hold this title. So, this also cannot be a named entity.

To further understand, let's just look at these NEs from a categories perspective:

Category	Examples of named entities
TIMEZONE	Asia/Kolkata, IST, UTC
LOCATION	Mumbai, Kolkata, Egypt
RIVERS	Ganga, Yamuna, Nile
COSMETICS	Maybelline Deep Coral Lipstick, LOreal Excellence Creme Hair Color
CURRENCY	100 bitcoins, 1.25 INR
DATE	17-Aug-2017, 19-Feb-2016

TIME	10:10 AM
PERSON	Satya Nadella, Jeff Weiner, Bill Gates

Using inbuilt NERs

Python NLTK has built-in support for **Named Entity Recognition (NER)**. In order to use this feature, first we need to recollect what we have done so far:

1. Break a large document into sentences.
2. Break the sentence into words (or tokens).
3. Identify the parts of speech in the sentence.
4. Extract chunks of consecutive words (non-overlapping) from the sentence.
5. Assign IOB tags to these words based on the chunking patterns.

The next logical step would be to further extend the algorithms to find out the named entities as a sixth step. So, we will basically be using data that is preprocessed until step 5 as part of this example.

We will be using `treebank` data to understand the NER process. Remember, the data is already pre-tagged in IOB format. Without the training process, none of the algorithms that we are seeing here are going to work. (So, there is no magic!)

In order to understand the importance of the training process, let's take up an example. Say, there is a need for the Archaeological department to figure out which of the famous places in India are being tweeted and mentioned in social networking websites in the Kannada Language.

Assuming that they have already got the data somewhere and it is in terabytes or even in petabytes, how do they find out all these names? This is where we need to take a sample dataset from the original input and do the training process to further use this trained data set to extract the named entities in Kannada.

Getting ready

You should have Python installed, along with the `nltk` library.

How to do it...

1. Open Atom editor (or you favorite programming editor).
2. Create a new file called `NER.py`.
3. Type the following source code:

```
NER.py
1 import nltk
2
3 def sampleNE():
4     sent = nltk.corpus.treebank.tagged_sents()[0]
5     print(nltk.ne_chunk(sent))
6
7 def sampleNE2():
8     sent = nltk.corpus.treebank.tagged_sents()[0]
9     print(nltk.ne_chunk(sent, binary=True))
10
11 if __name__ == '__main__':
12     sampleNE()
13     sampleNE2()
14
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:

A terminal window titled 'ch7 — -bash — 99x39' displays the output of a Python script named 'NER.py'. The script processes a sentence and identifies named entities. The output is as follows:

```
nltk $ python NER.py
(S
  (PERSON Pierre/NNP)
  (ORGANIZATION Vinken/NNP)
  ./,
  61/CD
  years/NN
  old/JJ
  ./,
  will/MD
  join/VB
  the/DT
  board/NN
  as/IN
  a/DT
  nonexecutive/JJ
  director/NN
  Nov./NNP
  29/CD
  ./.)
(S
  (NE Pierre/NNP Vinken/NNP)
  ./,
  61/CD
  years/NN
  old/JJ
  ./,
  will/MD
  join/VB
  the/DT
  board/NN
  as/IN
  a/DT
  nonexecutive/JJ
  director/NN
  Nov./NNP
  29/CD
  ./.)
nltk $
```

How it works...

The code looks so simple, right? However, all the algorithms are implemented in the `nltk` library. So, let's dig into how this simple program gives what we are looking for. This instruction imports the `nltk` library into the program:

```
import nltk
```

These three instructions define a new function called `sampleNE()`. We are importing the first tagged sentence from the `treebank` corpus and then passing it to the `nltk.ne_chunk()` function to extract the named entities. The output from this program includes all the named entities with their proper category:

```
def sampleNE():
    sent = nltk.corpus.treebank.tagged_sents()[0]
    print(nltk.ne_chunk(sent))
```

These three instructions define a new function called `sampleNE2()`. We are importing the first tagged sentence from the `treebank` corpus and then passing it to the `nltk.ne_chunk()` function to extract the named entities. The output from this program includes all the named entities without any proper category. This is helpful if the training dataset is not accurate enough to tag the named entities with the proper category such as person, organization, location, and so on:

```
def sampleNE2():
    sent = nltk.corpus.treebank.tagged_sents()[0]
    print(nltk.ne_chunk(sent, binary=True))
```

These three instructions will call the two sample functions that we have defined before and print the results on the screen.

```
if __name__ == '__main__':
    sampleNE()
    sampleNE2()
```

Creating, inversing, and using dictionaries

Python, as a general-purpose programming language, has support for many built-in data structures. Of those, one of the most powerful data structures are dictionaries. Before we jump into what dictionaries are, let's try to understand what these data structures are used for. Data structures, in short, help programmers to store, retrieve, and traverse through data that is stored in these structures. Each data structure has its own sets of behaviors and performance benefits that programmers should understand before selecting them for a given task at hand.

Let's get back to dictionaries. The basic use case of dictionaries can be explained with a simple example:

```
All the flights got delayed due to bad weather
```

We can use POS identification on the preceding sentence. But if someone were to ask what POS `flights` is in this sentence, we should have an efficient way to look for this word. This is where dictionaries come into play. They can be thought of as **one-to-one** mappings between data of interest. Again this one-to-one is at the highest level of abstraction of the data unit that we are talking about. If you are an expert programmer in Python, you know how to do **many-to-many** also. In this simple example, we need something like this:

```
flights -> Noun  
Weather -> Noun
```

Now let's answer a different question. Is it possible to print the list of all the words in the sentence that are nouns? Yes, for this too, we will learn how to use a Python dictionary.

Getting ready

You should have Python installed, along with the `nltk` library, in order to run this example.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `Dictionary.py`.

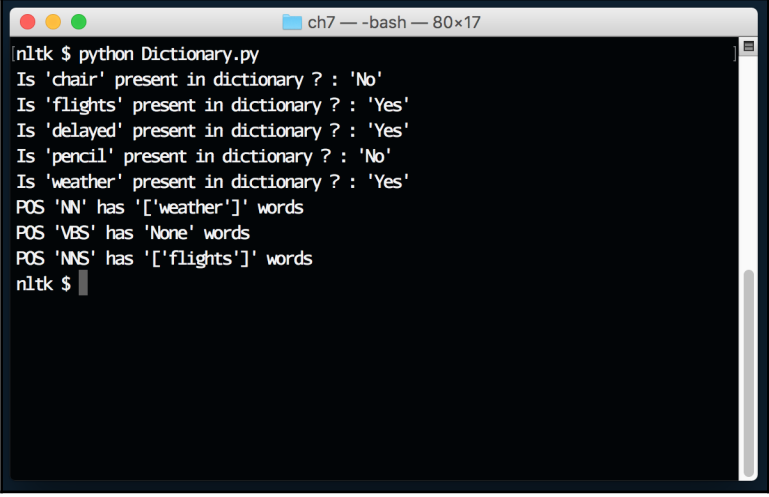
3. Type the following source code:

```

Dictionary.py
1  import nltk
2
3  class LearningDictionary():
4      def __init__(self, sentence):
5          self.words = nltk.word_tokenize(sentence)
6          self.tagged = nltk.pos_tag(self.words)
7          self.buildDictionary()
8          self.buildReverseDictionary()
9
10     def buildDictionary(self):
11         self.dictionary = {}
12         for (word, pos) in self.tagged:
13             self.dictionary[word] = pos
14
15     def buildReverseDictionary(self):
16         self.rdictionary = {}
17         for key in self.dictionary.keys():
18             value = self.dictionary[key]
19             if value not in self.rdictionary:
20                 self.rdictionary[value] = [key]
21             else:
22                 self.rdictionary[value].append(key)
23
24     def isWordPresent(self, word):
25         return 'Yes' if word in self.dictionary else 'No'
26
27
28     def getPOSForWord(self, word):
29         return self.dictionary[word] if word in self.dictionary else None
30
31     def getWordsForPOS(self, pos):
32         return self.rdictionary[pos] if pos in self.rdictionary else None
33
34
35     sentence = "All the flights got delayed due to bad weather"
36     learning = LearningDictionary(sentence)
37     words = ["chair", "flights", "delayed", "pencil", "weather"]
38     pos = ["NN", "VBS", "NNS"]
39     for word in words:
40         status = learning.isWordPresent(word)
41         print("Is '{}' present in dictionary ? : {}".format(word, status))
42         if status is True:
43             print("\tPOS For '{}' is {}".format(word,
44             * learning.getPOSForWord(word)))
45     for pword in pos:
46         print("POS '{}' has {} words".format(pword,
47         * learning.getWordsForPOS(pword)))

```


4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```
ch7 — -bash — 80x17
nltk $ python Dictionary.py
Is 'chair' present in dictionary ? : 'No'
Is 'flights' present in dictionary ? : 'Yes'
Is 'delayed' present in dictionary ? : 'Yes'
Is 'pencil' present in dictionary ? : 'No'
Is 'weather' present in dictionary ? : 'Yes'
POS 'NN' has '['weather']' words
POS 'VBS' has 'None' words
POS 'NNS' has '['flights']' words
nltk $
```

How it works...

Now, let's understand more about dictionaries by going through the instructions we have written so far. We are importing the `nltk` library into the program:

```
import nltk
```

We are defining a new class called `LearningDictionary`:

```
class LearningDictionary():
```

We are creating a constructor for `LearningDictionary` that takes sentence text as an argument:

```
def __init__(self, sentence):
```

This instruction breaks the sentence into words using the `nltk.word_tokenize()` function and saves the result in the class member `words`:

```
self.words = nltk.word_tokenize(sentence)
```

This instruction identifies the POS for words and saves the result in the class member `tagged`:

```
self.tagged = nltk.pos_tag(self.words)
```

This instruction invokes the `buildDictionary()` function that is defined in the class:

```
self.buildDictionary()
```

This instruction invokes the `buildReverseDictionary()` function that is defined in the class:

```
self.buildReverseDictionary()
```

This instruction defines a new class member function called `buildDictionary()`:

```
def buildDictionary(self):
```

This instruction initializes a empty dictionary variable in the class. These two instructions iterate over all the tagged pos list elements and then assign each word to the dictionary as key and the POS as value of the key:

```
self.dictionary = {}  
for (word, pos) in self.tagged:  
    self.dictionary[word] = pos
```

This instruction defines another class member function called `buildReverseDictionary()`:

```
def buildReverseDictionary(self):
```

This instruction initializes an empty dictionary to a class member, `rdictionary`:

```
self.rdictionary = {}
```

This instruction iterates over all the dictionary keys and puts the key of dictionary into a local variable called `key`:

```
for key in self.dictionary.keys():
```

This instruction extracts the value (POS) of the given key (word) and stores it in a local variable called `value`:

```
value = self.dictionary[key]
```

These four instructions check whether a given key (word) is already in the reverse dictionary variable (`rdictionary`). If it is, then we append the currently found word to the list. If the word is not found, then we create a new list of size one with the current word as the member:

```
if value not in self.rdictionary:
    self.rdictionary[value] = [key]
else:
    self.rdictionary[value].append(key)
```

This function returns Yes or No depending on whether a given word is found in the dictionary:

```
def isWordPresent(self, word):
    return 'Yes' if word in self.dictionary else 'No'
```

This function returns the POS for the given word by looking into dictionary. If the value is not found, a special value of `None` is returned:

```
def getPOSForWord(self, word):
    return self.dictionary[word] if word in self.dictionary else None
```

These two instructions define a function that returns all the words in the sentence with a given POS by looking into `rdictionary` (reverse dictionary). If the POS is not found, a special value of `None` is returned:

```
def getWordsForPOS(self, pos):  
    return self.rdictionary[pos] if pos in self.rdictionary else None
```

We define a variable called `sentence`, which stores the string that we are interested in parsing:

```
sentence = "All the flights got delayed due to bad weather"
```

Initialize the `LearningDictionary()` class with `sentence` as a parameter. Once the class object is created, it is assigned to the learning variable:

```
learning = LearningDictionary(sentence)
```

We create a list of words that we are interested in knowing the POS of. If you see carefully, we have included a few words that are not in the sentence:

```
words = ["chair", "flights", "delayed", "pencil", "weather"]
```

We create a list of `pos` for which we are interested in seeing the words that belong to these POS classifications:

```
pos = ["NN", "VBS", "NNS"]
```

These instructions iterate over all the words, take one word at a time, check whether the word is in the dictionary by calling the `isWordPresent()` function of the object, and then print its status. If the word is present in the dictionary, then we print the POS for the word:

```
for word in words:  
    status = learning.isWordPresent(word)  
    print("Is '{}' present in dictionary ? : {}".format(word, status))  
    if status is True:  
        print("\tPOS For '{}' is {}".format(word,  
learning.getPOSForWord(word)))
```

In these instructions, we iterate over all the `pos`. We take one word at a time and then print the words that are in this POS using the `getWordsForPOS()` function.

```
for pword in pos:
    print("POS '{}' has '{}' words".format(pword,
        learning.getWordsForPOS(pword)))
```

Choosing the feature set

Features are one of the most powerful components of `nltk` library. They represent clues within the language for easy tagging of the data that we are dealing with. In Python terminology, features are expressed as dictionaries, where the keys are the labels and the values are the clues extracted from input data.

Let's say we are dealing with some transport department data and we are interested in finding out whether a given vehicle number belongs to the Government of Karnataka or not. Right now we have no clue about the data we are dealing with. So how can we tag the given numbers accurately?

Let's try to learn how the vehicle numbers give some clues about what they mean:

Vehicle number	Clues about the pattern
KA-[0-9]{2} [0-9]{2}	Normal vehicle number
KA-[0-9]{2}-F	KSRTC, BMTC vehicles
KA-[0-9]{2}-G	Government vehicles

Using these clues (features), let's try to come up with a simple program that can tell us the classification of a given input number.

Getting ready

You should have Python installed, along with the `nltk` library.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `Features.py`.

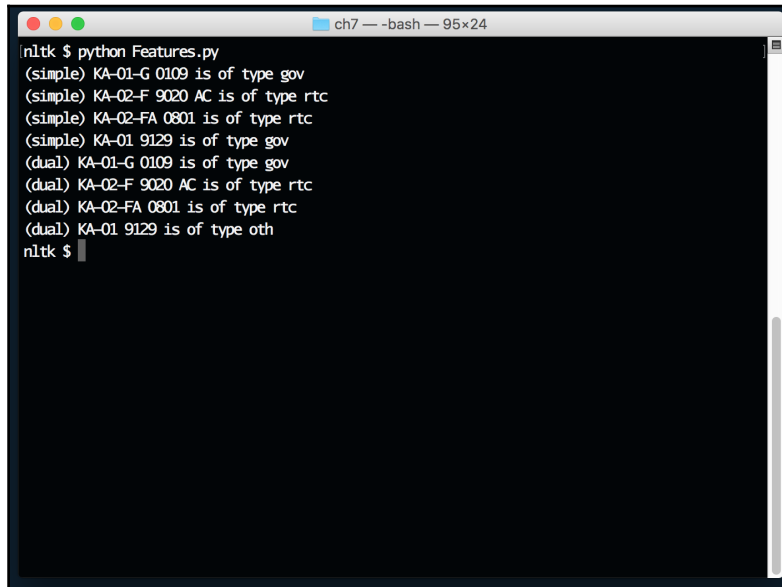
3. Type the following source code:

```

Features.py
1  import nltk
2  import random
3
4  sampledata = [
5      ('KA-01-F 1034 A', 'rtc'),
6      ('KA-02-F 1030 B', 'rtc'),
7      ('KA-03-FA 1200 C', 'rtc'),
8      ('KA-01-G 0001 A', 'gov'),
9      ('KA-02-G 1004 A', 'gov'),
10     ('KA-03-G 0204 A', 'gov'),
11     ('KA-04-G 9230 A', 'gov'),
12     ('KA-27 1290', 'oth')
13 ]
14 random.shuffle(sampledata)
15 testdata = [
16     'KA-01-G 0109',
17     'KA-02-F 9020 AC',
18     'KA-02-FA 0801',
19     'KA-01 9129'
20 ]
21 def learnSimpleFeatures():
22     def vehicleNumberFeature(vnumber):
23         return {'vehicle_class': vnumber[6]}
24     featuresets = [(vehicleNumberFeature(vn), cls) for (vn, cls) in sampledata]
25     classifier = nltk.NaiveBayesClassifier.train(featuresets)
26     for num in testdata:
27         feature = vehicleNumberFeature(num)
28         print("(simple) %s is of type %s" %(num, classifier.classify(feature)))
29
30 def learnFeatures():
31     def vehicleNumberFeature(vnumber):
32         return {
33             'vehicle_class': vnumber[6],
34             'vehicle_prev': vnumber[5]
35         }
36     featuresets = [(vehicleNumberFeature(vn), cls) for (vn, cls) in sampledata]
37     classifier = nltk.NaiveBayesClassifier.train(featuresets)
38     for num in testdata:
39         feature = vehicleNumberFeature(num)
40         print("(dual) %s is of type %s" %(num, classifier.classify(feature)))
41
42 learnSimpleFeatures()
43 learnFeatures()
44

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:

A terminal window titled 'ch7 -- -bash -- 95x24' displays the output of a Python script. The script is 'Features.py' and it outputs a series of classification results for different vehicle numbers. The results are as follows:

```
nltk $ python Features.py
(simple) KA-01-G 0109 is of type gov
(simple) KA-02-F 9020 AC is of type rtc
(simple) KA-02-FA 0801 is of type rtc
(simple) KA-01 9129 is of type gov
(dual) KA-01-G 0109 is of type gov
(dual) KA-02-F 9020 AC is of type rtc
(dual) KA-02-FA 0801 is of type rtc
(dual) KA-01 9129 is of type oth
nltk $
```

How it works...

Now, let's see what our program does. These two instructions import the `nltk` and `random` libraries into the current program:

```
import nltk
import random
```

We are defining a list of Python tuples, where the first element in the tuple is the vehicle number and the second element is the predefined label that is applied to the number.

These instructions define that all the numbers are classified into three labels—`rtc`, `gov`, and `oth`:

```
sampladata = [
    ('KA-01-F 1034 A', 'rtc'),
    ('KA-02-F 1030 B', 'rtc'),
    ('KA-03-FA 1200 C', 'rtc'),
    ('KA-01-G 0001 A', 'gov'),
```

```
('KA-02-G 1004 A', 'gov'),
('KA-03-G 0204 A', 'gov'),
('KA-04-G 9230 A', 'gov'),
('KA-27 1290', 'oth')
]
```

This instruction shuffles all of the data in the `sampledata` list to make sure that the algorithm is not biased by the order of elements in the input sequence:

```
random.shuffle(sampledata)
```

These are the test vehicle numbers for which we are interested in finding the category:

```
testdata = [
    'KA-01-G 0109',
    'KA-02-F 9020 AC',
    'KA-02-FA 0801',
    'KA-01 9129'
]
```

This instruction defines a new function called `learnSimpleFeatures`:

```
def learnSimpleFeatures():
```

These instructions define a new function, `vehicleNumberFeature()`, which takes the vehicle number and returns the seventh character in the that number. The return type is dictionary:

```
def vehicleNumberFeature(vnumber):
    return {'vehicle_class': vnumber[6]}
```

This instruction creates a list of feature tuples, where the first member in the tuple is feature dictionary and the second member in tuple is the label of the data. After this instruction, the input vehicle numbers in `sampledata` are no longer visible. This is one of the key things to remember:

```
featuresets = [(vehicleNumberFeature(vn), cls) for (vn, cls) in sampledata]
```

This instruction trains `NaiveBayesClassifier` with the feature dictionary and the labels that are applied to `featuresets`. The result is available in the classifier object, which we will use further:

```
classifier = nltk.NaiveBayesClassifier.train(featuresets)
```


These instructions iterate over the test data and then print the label of the data from the classification done using `vehicleNumberFeature`. Observe the output carefully. You will see that the feature extraction function that we have written does not perform well in labeling the numbers correctly:

```
for num in testdata:
    feature = vehicleNumberFeature(num)
    print("(simple) %s is of type %s" %(num, classifier.classify(feature)))
```

This instruction defines a new function called `learnFeatures`:

```
def learnFeatures():
```

These instructions define a new function called `vehicleNumberFeature` that returns the feature dictionary with two keys. One key, `vehicle_class`, returns the character at position 6 in the string, and `vehicle_prev` has the character at position 5. These kinds of clues are very important to make sure we eliminate bad labeling of data:

```
def vehicleNumberFeature(vnumber):
    return {
        'vehicle_class': vnumber[6],
        'vehicle_prev': vnumber[5]
    }
```

This instruction creates a list of `featuresets` and input labels by iterating over of all the input trained data. As before, the original input vehicle numbers are no longer present here:

```
featuresets = [(vehicleNumberFeature(vn), cls) for (vn, cls) in sampledata]
```

This instruction creates a `NaiveBayesClassifier.train()` function on `featuresets` and returns the object for future use:

```
classifier = nltk.NaiveBayesClassifier.train(featuresets)
```

These instructions loop through the `testdata` and print the classification of the input vehicle number based on the trained dataset. Here, if you observe carefully, the false-positive is not there:

```
for num in testdata:
    feature = vehicleNumberFeature(num)
    print("(dual) %s is of type %s" %(num, classifier.classify(feature)))
```

Invoke both the functions and print the results on the screen.

```
learnSimpleFeatures()  
learnFeatures()
```

If we observe carefully, we realize that the first function's results have one false positive, where it cannot identify the `gov` vehicle. This is where the second function performs well, as it has more features that improve accuracy.

Segmenting sentences using classification



A natural language that supports question marks (?), full stops (.), and exclamations (!) poses a challenge to us in identifying whether a statement has ended or it still continues after the punctuation characters.

This is one of the classic problems to solve.

In order to solve this problem, let's find out the features (or clues) that we can leverage to come up with a classifier and then use the classifier to extract sentences in large text.



If we encounter a punctuation mark like `.` then it ends a sentence. If we encounter a punctuation mark like `.` and the next word's first letter is a capital letter, then it ends a sentence.

Let's try to write a simple classifier using these two features to mark sentences.

Getting ready

You should have Python installed along with `nltk` library.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new File called `Segmentation.py`.

3. Type the following source code:

```

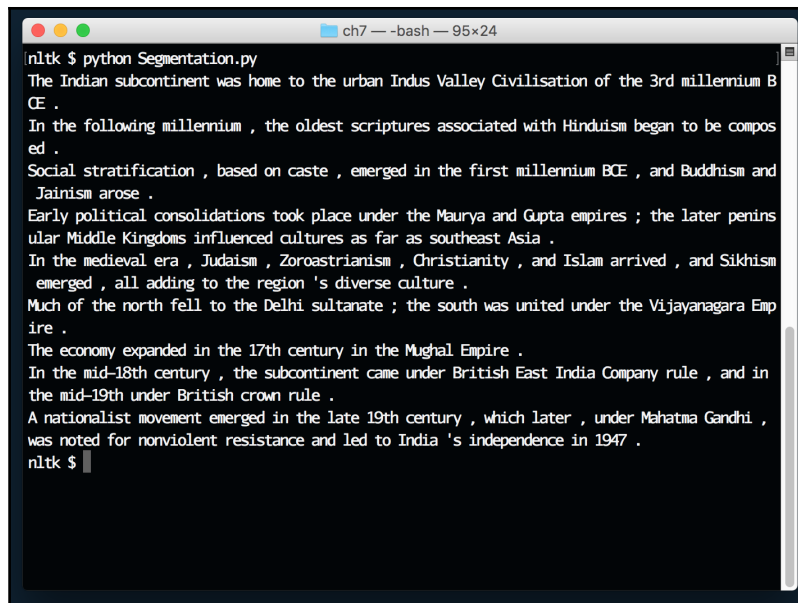
Segmentation.py
1 import nltk
2 def featureExtractor(words, i):
3     return ({'current-word': words[i], 'next-is-upper':
4         words[i+1][0].isupper(), words[i+1][0].isupper()})
5 def getFeaturesets(sentence):
6     words = nltk.word_tokenize(sentence)
7     featuresets = [featureExtractor(words, i) for i in range(1, len(words) - 1)
8         if words[i] == '.']
9     return featuresets
10 def segmentTextAndPrintSentences(data):
11     words = nltk.word_tokenize(data)
12     for i in range(0, len(words) - 1):
13         if words[i] == '.':
14             if classifier.classify(featureExtractor(words, i)[0]) == True:
15                 print(".")
16             else:
17                 print(words[i], end='')
18         else:
19             print("{} ".format(words[i]), end='')
20     print(words[-1])
21 # copied the text from https://en.wikipedia.org/wiki/India
22 traindata = "India, officially the Republic of India (Bhārat Gaṇarājya),[e] is
23 a country in South Asia. it is the seventh-largest country by area, the
24 second-most populous country (with over 1.2 billion people), and the most
25 populous democracy in the world. It is bounded by the Indian Ocean on the
26 south, the Arabian Sea on the southwest, and the Bay of Bengal on the
27 southeast. It shares land borders with Pakistan to the west;[f] China, Nepal,
28 and Bhutan to the northeast; and Myanmar (Burma) and Bangladesh to the east. In
29 the Indian Ocean, India is in the vicinity of Sri Lanka and the Maldives.
30 India's Andaman and Nicobar Islands share a maritime border with Thailand and
31 Indonesia."
32 testdata = "The Indian subcontinent was home to the urban Indus Valley
33 Civilisation of the 3rd millennium BCE. In the following millennium, the oldest
34 scriptures associated with Hinduism began to be composed. Social
35 stratification, based on caste, emerged in the first millennium BCE, and
36 Buddhism and Jainism arose. Early political consolidations took place under the
37 Maurya and Gupta empires; the later peninsular Middle Kingdoms influenced
38 cultures as far as southeast Asia. In the medieval era, Judaism,
39 Zoroastrianism, Christianity, and Islam arrived, and Sikhism emerged, all
40 adding to the region's diverse culture. Much of the north fell to the Delhi
41 sultanate; the south was united under the Vijayanagara Empire. The economy
42 expanded in the 17th century in the Mughal Empire. In the mid-18th century, the
43 subcontinent came under British East India Company rule, and in the mid-19th
44 under British crown rule. A nationalist movement emerged in the late 19th
45 century, which later, under Mahatma Gandhi, was noted for nonviolent resistance
46 and led to India's independence in 1947."
47
48 traindataset = getFeaturesets(traindata)
49 classifier = nltk.NaiveBayesClassifier.train(traindataset)
50 segmentTextAndPrintSentences(testdata)
51
52

```

4. Save the file.

5. Run the program using the Python interpreter.

6. You will see the following output:

A terminal window titled 'ch7 -- -bash -- 95x24' displays the output of a Python script. The script uses NLTK to process a paragraph about Indian history. The output shows the text with sentence boundaries identified by periods. The text is: 'The Indian subcontinent was home to the urban Indus Valley Civilisation of the 3rd millennium B CE . In the following millennium , the oldest scriptures associated with Hinduism began to be composed . Social stratification , based on caste , emerged in the first millennium BCE , and Buddhism and Jainism arose . Early political consolidations took place under the Maurya and Gupta empires ; the later peninsular Middle Kingdoms influenced cultures as far as southeast Asia . In the medieval era , Judaism , Zoroastrianism , Christianity , and Islam arrived , and Sikhism emerged , all adding to the region 's diverse culture . Much of the north fell to the Delhi sultanate ; the south was united under the Vijayanagara Empire . The economy expanded in the 17th century in the Mughal Empire . In the mid-18th century , the subcontinent came under British East India Company rule , and in the mid-19th under British crown rule . A nationalist movement emerged in the late 19th century , which later , under Mahatma Gandhi , was noted for nonviolent resistance and led to India 's independence in 1947 .'. The terminal prompt 'nlk \$' is visible at the bottom.

```
nlk $ python Segmentation.py
The Indian subcontinent was home to the urban Indus Valley Civilisation of the 3rd millennium B
CE .
In the following millennium , the oldest scriptures associated with Hinduism began to be compos
ed .
Social stratification , based on caste , emerged in the first millennium BCE , and Buddhism and
Jainism arose .
Early political consolidations took place under the Maurya and Gupta empires ; the later penins
ular Middle Kingdoms influenced cultures as far as southeast Asia .
In the medieval era , Judaism , Zoroastrianism , Christianity , and Islam arrived , and Sikhism
emerged , all adding to the region 's diverse culture .
Much of the north fell to the Delhi sultanate ; the south was united under the Vijayanagara Emp
ire .
The economy expanded in the 17th century in the Mughal Empire .
In the mid-18th century , the subcontinent came under British East India Company rule , and in
the mid-19th under British crown rule .
A nationalist movement emerged in the late 19th century , which later , under Mahatma Gandhi ,
was noted for nonviolent resistance and led to India 's independence in 1947 .
nlk $
```

How it works...

This instruction imports the `nltk` library into the program:

```
import nltk
```

This function defines a modified feature extractor that returns a tuple containing the dictionary of the features and `True` or `False` to tell whether this feature indicates a sentence boundary or not:

```
def featureExtractor(words, i):
    return ({'current-word': words[i], 'next-is-upper':
words[i+1][0].isupper()}, words[i+1][0].isupper())
```

This function takes a sentence as input and returns a list of featuresets that is a list of tuples, with the feature dictionary and True or False:

```
def getFeaturesets(sentence):
    words = nltk.word_tokenize(sentence)
    featuresets = [featureExtractor(words, i) for i in range(1, len(words) -
1) if words[i] == '.']
    return featuresets
```

This function takes the input text, breaks it into words, and then traverses through each word in the list. Once it encounters a full stop, it calls `classifier` to conclude whether it has encountered a sentence end. If the `classifier` returns True, then the sentence is found and we move on to the next word in the input. The process is repeated for all words in the input:

```
def segmentTextAndPrintSentences(data):
    words = nltk.word_tokenize(data)
    for i in range(0, len(words) - 1):
        if words[i] == '.':
            if classifier.classify(featureExtractor(words, i)[0]) == True:
                print(".")
            else:
                print(words[i], end='')
        else:
            print("{} ".format(words[i]), end='')
    print(words[-1])
```

These instructions define a few variables for training and evaluation of our classifier:

```
# copied the text from https://en.wikipedia.org/wiki/India
traindata = "India, officially the Republic of India (Bhārat Gaṇarājya),[e]
is a country in South Asia. it is the seventh-largest country by area, the
second-most populous country (with over 1.2 billion people), and the most
populous democracy in the world. It is bounded by the Indian Ocean on the
south, the Arabian Sea on the southwest, and the Bay of Bengal on the
southeast. It shares land borders with Pakistan to the west;[f] China,
Nepal, and Bhutan to the northeast; and Myanmar (Burma) and Bangladesh to
the east. In the Indian Ocean, India is in the vicinity of Sri Lanka and
the Maldives. India's Andaman and Nicobar Islands share a maritime border
with Thailand and Indonesia."
```

```
testdata = "The Indian subcontinent was home to the urban Indus Valley
Civilisation of the 3rd millennium BCE. In the following millennium, the
oldest scriptures associated with Hinduism began to be composed. Social
stratification, based on caste, emerged in the first millennium BCE, and
Buddhism and Jainism arose. Early political consolidations took place under
the Maurya and Gupta empires; the later peninsular Middle Kingdoms
```

influenced cultures as far as southeast Asia. In the medieval era, Judaism, Zoroastrianism, Christianity, and Islam arrived, and Sikhism emerged, all adding to the region's diverse culture. Much of the north fell to the Delhi sultanate; the south was united under the Vijayanagara Empire. The economy expanded in the 17th century in the Mughal Empire. In the mid-18th century, the subcontinent came under British East India Company rule, and in the mid-19th under British crown rule. A nationalist movement emerged in the late 19th century, which later, under Mahatma Gandhi, was noted for nonviolent resistance and led to India's independence in 1947."

Extract all the features from the `traindata` variable and store it in `traindataset`:

```
traindataset = getFeaturesets(traindata)
```

Train the `NaiveBayesClassifier` on `traindataset` to get `classifier` as object:

```
classifier = nltk.NaiveBayesClassifier.train(traindataset)
```

Invoke the function on `testdata` and print all the found sentences as output on the screen:

```
segmentTextAndPrintSentences(testdata)
```

Classifying documents

In this recipe, we will learn how to write a classifier that can be used to classify documents. In our case, we will classify **rich site summary (RSS)** feeds. The list of categories is known ahead of time, which is important for the classification task.

In this information age, there are vast amounts of text available. Its humanly impossible for us to properly categorize all information for further consumption. This is where categorization algorithms help us to properly categorize the newer sets of documents that are being produced based on the training given on sample data.

Getting ready

You should have Python installed, along with the `nltk` and `feedparser` libraries.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `DocumentClassify.py`.

3. Type the following source code:

```

DocumentClassify.py
1 import nltk
2 import random
3 import feedparser
4
5 urls = {
6     'mlb': 'https://sports.yahoo.com/mlb/rss.xml',
7     'nfl': 'https://sports.yahoo.com/nfl/rss.xml',
8 }
9
10 feedmap = {}
11 stopwords = nltk.corpus.stopwords.words('english')
12
13 def featureExtractor(words):
14     features = {}
15     for word in words:
16         if word not in stopwords:
17             features["word({})".format(word)] = True
18     return features
19
20 sentences = []
21
22 for category in urls.keys():
23     feedmap[category] = feedparser.parse(urls[category])
24     print("downloading {}".format(urls[category]))
25     for entry in feedmap[category]['entries']:
26         data = entry['summary']
27         words = data.split()
28         sentences.append((category, words))
29
30 featuresets = [(featureExtractor(words), category) for category, words in
31                sentences]
32 random.shuffle(featuresets)
33
34 total = len(featuresets)
35 off = int(total/2)
36 trainset = featuresets[off:]
37 testset = featuresets[:off]
38
39 classifier = nltk.NaiveBayesClassifier.train(trainset)
40
41 print(nltk.classify.accuracy(classifier, testset))
42
43 classifier.show_most_informative_features(5)
44 for (i, entry) in enumerate(feedmap['nfl']['entries']):
45     if i < 4:
46         features = featureExtractor(entry['title'].split())
47         category = classifier.classify(features)
48         print('{} -> {}'.format(category, entry['summary']))
49

```

4. Save the file.

5. Run the program using the Python interpreter.
6. You will see this output:

```

ch7 - bash - 95x24
nlTK $ python DocumentClassify.py
downloading https://sports.yahoo.com/mlb/rss.xml
downloading https://sports.yahoo.com/nfl/rss.xml
0.8775510204081632
Most Informative Features
      word(Association) = True      mlb : nfl   =      2.8 : 1.0
      word(second) = True          nfl : mlb   =      2.5 : 1.0
      word(New) = True             mlb : nfl   =      2.2 : 1.0
      word(He) = True              mlb : nfl   =      2.2 : 1.0
      word(first) = True           mlb : nfl   =      2.2 : 1.0

nfl -> Monday night's ugly loss to the Lions showed that Brett Hundley can't possibly replace A
aron Rodgers. In case you didn't know that already.
nfl -> Green Bay, Wis. - It was midway through the third quarter and the Green Bay Packers stil
l had a pulse. Down 14 points and at midfield, the Packers faced a fourth-and-2. Going for it w
as an easy choice and the play call was a good one against the man-coverage look the Lions were
showing. Wide receiver
nfl -> Green Bay, Wis. - This is exactly what the Lions needed from Matthew Stafford, and exact
ly when they needed it. They couldn't lose this one, not with their season teetering, not with
the Packers missing Aaron Rodgers. They couldn't squander this matchup on Monday, and Stafford
played as if he
nfl -> The Lions went into Lambeau and wrecked the Packers. The Detroit News team offers their
instant analysis from the win.
nlTK $

```

How it works...

Let's see how this document classification works. Importing three libraries into the program:

```

import nltk
import random
import feedparser

```

This instruction defines a new dictionary with two RSS feeds pointing to Yahoo! sports. They are pre-categorized. The reason we have selected these RSS feeds is that data is readily available and categorized for our example:

```

urls = {
    'mlb': 'https://sports.yahoo.com/mlb/rss.xml',
    'nfl': 'https://sports.yahoo.com/nfl/rss.xml',
}

```


Initializing the empty dictionary variable `feedmap` to keep the list of RSS feeds in memory until the program terminates:

```
feedmap = {}
```

Getting the list of stopwords in English and storing it in the `stopwords` variable:

```
stopwords = nltk.corpus.stopwords.words('english')
```

This function, `featureExtractor()`, takes list of words and then adds them to the dictionary, where each key is the word and the value is `True`. The dictionary is returned, which are the features for the given input words:

```
def featureExtractor(words):  
    features = {}  
    for word in words:  
        if word not in stopwords:  
            features["word({})".format(word)] = True  
    return features
```

Empty list to store all the correctly labeled sentences:

```
sentences = []
```

Iterate over all the `keys()` of the dictionary called `urls` and store the key in a variable called `category`:

```
for category in urls.keys():
```

Download one feed and store the result in the `feedmap[category]` variable using the `parse()` function from the `feedparser` module:

```
feedmap[category] = feedparser.parse(urls[category])
```

Display the `url` that is being downloaded on the screen, using Python's built-in the `print` function:

```
print("downloading {}".format(urls[category]))
```

Iterate over all the RSS entries and store the current entry in a variable called `entry` variable:

```
for entry in feedmap[category]['entries']:
```

Take the `summary` (news text) of the RSS feed item into the `data` variable:

```
data = entry['summary']
```

We break `summary` into words based on space so that we can pass these to `nltk` for feature extraction:

```
words = data.split()
```

Store all words in the current RSS feed item, along with `category` it belongs to, in a tuple:

```
sentences.append((category, words))
```

Extract all the features of `sentences` and store them in the variable `featuresets`. Later, do `shuffle()` on this array so that all the elements in the list are randomized for the algorithm:

```
featuresets = [(featureExtractor(words), category) for category, words in sentences]
random.shuffle(featuresets)
```

Create two datasets, one `trainset` and the other `testset`, for our analysis:

```
total = len(featuresets)
off = int(total/2)
trainset = featuresets[off:]
testset = featuresets[:off]
```

Create a classifier using the `trainset` data by using the `NaiveBayesClassifier` module's `train()` function:

```
classifier = nltk.NaiveBayesClassifier.train(trainset)
```

Print the accuracy of `classifier` using `testset`:

```
print(nltk.classify.accuracy(classifier, testset))
```

Print the informative features about this data using the built-in function of classifier:

```
classifier.show_most_informative_features(5)
```

Take four sample entries from the nfl RSS item. Try to tag the document based on title (remember, we have classified them based on summary):

```
for (i, entry) in enumerate(feedmap['nfl']['entries']):
    if i < 4:
        features = featureExtractor(entry['title'].split())
        category = classifier.classify(features)
        print('{} -> {}'.format(category, entry['title']))
```

Writing a POS tagger with context

In previous recipes, we have written regular-expression-based POS taggers that leverage word suffixes such as *ed*, *ing*, and so on to check whether the word is of a given POS or not. In the English language, the same word can play a dual role depending on the context in which it is used.

For example, the word *address* can be both noun and verb depending on the context:

```
"What is your address when you're in Bangalore?"
"the president's address on the state of the economy."
```

Let's try to write a program that leverages the feature extraction concept to find the POS of the words in the sentence.

Getting ready

You should have Python installed, along with `nltk`.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `ContextTagger.py`.

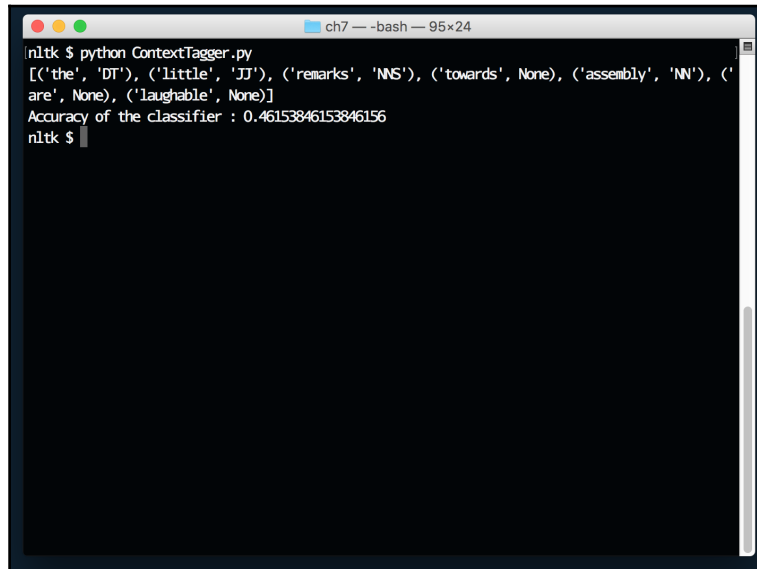
3. Type the following source code:

```

ContextTagger.py
1 import nltk
2 sentences = [
3     "What is your address when you're in Bangalore?",
4     "the president's address on the state of the economy.",
5     "He addressed his remarks to the lawyers in the audience.",
6     "In order to address an assembly, we should be ready",
7     "He laughed inwardly at the scene.",
8     "After all the advance publicity, the prizefight turned out to be a laugh.",
9     "We can learn to laugh a little at even our most serious foibles."
10 ]
11 def getSentenceWords():
12     sentwords = []
13     for sentence in sentences:
14         words = nltk.pos_tag(nltk.word_tokenize(sentence))
15         sentwords.append(words)
16     return sentwords
17 def noContextTagger():
18     tagger = nltk.UnigramTagger(getSentenceWords())
19     print(tagger.tag('the little remarks towards assembly are
20     * laughable'.split()))
21 def withContextTagger():
22     def wordFeatures(words, wordPosInSentence):
23         # extract all the ing forms etc
24         endFeatures = {
25             'last(1)': words[wordPosInSentence][-1],
26             'last(2)': words[wordPosInSentence][-2:],
27             'last(3)': words[wordPosInSentence][-3:],
28         }
29         # use previous word to determine if the current word is verb or noun
30         if wordPosInSentence > 1:
31             endFeatures['prev'] = words[wordPosInSentence - 1]
32         else:
33             endFeatures['prev'] = '|NONE|'
34         return endFeatures
35     allsentences = getSentenceWords()
36     featureddata = []
37     for sentence in allsentences:
38         untaggedSentence = nltk.tag.untag(sentence)
39         featuredsentence = [(wordFeatures(untaggedSentence, index), tag) for
40         * index, (word, tag) in enumerate(sentence)]
41         featureddata.extend(featuredsentence)
42     breakup = int(len(featureddata) * 0.5)
43     traindata = featureddata[breakup:]
44     testdata = featureddata[:breakup]
45     classifier = nltk.NaiveBayesClassifier.train(traindata)
46     print("Accuracy of the classifier :
47     * {}".format(nltk.classify.accuracy(classifier, testdata)))
48
49 noContextTagger()
50 withContextTagger()

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```
nltk $ python ContextTagger.py
[('the', 'DT'), ('little', 'JJ'), ('remarks', 'NNS'), ('towards', None), ('assembly', 'NN'), ('are', None), ('laughable', None)]
Accuracy of the classifier : 0.46153846153846156
nltk $
```

How it works...

Let's see how the current program works. This instruction imports the `nltk` library into the program:

```
import nltk
```

Some sample strings that indicate the dual behavior of the words, address, laugh:

```
sentences = [
    "What is your address when you're in Bangalore?",
    "the president's address on the state of the economy.",
    "He addressed his remarks to the lawyers in the audience.",
    "In order to address an assembly, we should be ready",
    "He laughed inwardly at the scene.",
    "After all the advance publicity, the prizefight turned out to be a laugh.",
    "We can learn to laugh a little at even our most serious foibles."
]
```

This function takes sentence strings and returns a list of lists, where the inner lists contain the words along with their POS tags:

```
def getSentenceWords():
    sentwords = []
    for sentence in sentences:
        words = nltk.pos_tag(nltk.word_tokenize(sentence))
        sentwords.append(words)
    return sentwords
```

In order to set up a baseline and see how bad the tagging can be, this function explains how `UnigramTagger` can be used to print the POS of the words just by looking at the current word. We are feeding the sample text to it as learning. This tagger performs very badly when compared to the built-in tagger that `nltk` comes with. But this is just for our understanding:

```
def noContextTagger():
    tagger = nltk.UnigramTagger(getSentenceWords())
    print(tagger.tag('the little remarks towards assembly are
laughable'.split()))
```

Defining a new function called `withContextTagger()`:

```
def withContextTagger():
```

This function does feature extraction on a given set of words and returns a dictionary of the last three characters of the current word and previous word information:

```
def wordFeatures(words, wordPosInSentence):
    # extract all the ing forms etc
    endFeatures = {
        'last(1)': words[wordPosInSentence][-1],
        'last(2)': words[wordPosInSentence][-2:],
        'last(3)': words[wordPosInSentence][-3:],
    }
    # use previous word to determine if the current word is verb or noun
    if wordPosInSentence > 1:
        endFeatures['prev'] = words[wordPosInSentence - 1]
    else:
        endFeatures['prev'] = '|NONE|'
    return endFeatures
```

We are building a `featuredata` list. It contains tuples of `featurelist` and `tag` members, which we will use to classify using `NaiveBayesClassifier`:

```
allsentences = getSentenceWords()
featureddata = []
for sentence in allsentences:
    untaggedSentence = nltk.tag.untag(sentence)
    featuredsentence = [(wordFeatures(untaggedSentence, index), tag) for
index, (word, tag) in enumerate(sentence)]
    featureddata.extend(featuredsentence)
```

We take 50% for training and 50% of the feature extracted data to test our classifier:

```
breakup = int(len(featureddata) * 0.5)
traindata = featureddata[breakup:]
testdata = featureddata[:breakup]
```

This instruction creates classifier using the training data:

```
classifier = nltk.NaiveBayesClassifier.train(traindata)
```

This instruction prints the accuracy of the classifier using `testdata`:

```
print("Accuracy of the classifier :
{}".format(nltk.classify.accuracy(classifier, testdata)))
```

These two functions print the results of two preceding functions' computations.

```
noContextTagger()
withContextTagger()
```

8

Advanced NLP Recipes

In this chapter, we will go through the following recipes:

- Creating an NLP pipeline
- Solving the text similarity problem
- Identifying topics
- Summarizing text
- Resolving anaphora
- Disambiguating word sense
- Perform sentiment analysis
- Exploring advanced sentiment analysis
- Creating a conversational assistant or chatbot

Introduction

So far, we have seen how to process input text, identify parts of speech, and extract important information (named entities). We've learned a few computer science concepts also, such as grammars, parsers, and so on. In this chapter, we will dig deeper into advanced topics in **natural language processing (NLP)**, which need several techniques to properly understand and solve them.

Creating an NLP pipeline

In computing, a pipeline can be thought of as a multi-phase data flow system where the output from one component is fed to the input of another component.

These are the things that happen in a pipeline:

- Data is flowing all the time from one component to another
- The component is a black box that should worry about the input data and output data

A well-defined pipeline takes care of the following things.

- The input format of the data that is flowing through each of the components
- The output format of the data that is coming out of each of the components
- Making sure that data flow is controlled between components by adjusting the velocity of data inflow and outflow

For example, if you are familiar with Unix/Linux systems and have some exposure to working on a shell, you'd have seen the `|` operator, which is the shell's abstraction of a data pipe. We can leverage the `|` operator to build pipelines in the Unix shell.

Let's take an example in Unix (for a quick understanding): how do I find the number of files in a given directory ?

To solve this, we need the following things:

- We need a component (or a command in the Unix context) that reads the directory and lists all the files in it
- We need another component (or a command in the Unix context) that reads the lines and prints the count of lines

So, we have the solutions to these two requirements. Which are :

- The `ls` command
- The `wc` command

If we can build a pipeline where we take the output from `ls` and feed it to `wc`, we are done.

In terms of Unix commands, `ls -l | wc -l` is a simple pipeline that counts the files in a directory.

With this knowledge, let's get back to the NLP pipeline requirements:

- Input data acquisition
- Breaking the input data into words
- Identifying the POS of words in the input data
- Extracting the named entities from the words
- Identifying the relationships between named entities

In this recipe, let's try to build the simplest possible pipeline; it acquires data from a remote RSS feed and then prints the identified named entities in each document.

Getting ready

You should have Python installed, along with the `nltk`, `queue`, `feedparser`, and `uuid` libraries.

How to do it...

1. Open Atom editor (or your favorite programming editor).
2. Create a new file called `PipelineQ.py`.

3. Type the following source code:

```

PipelineQ.py

1 import nltk
2 import threading
3 import queue
4 import feedparser
5 import uuid
6
7 threads = []
8 queues = [queue.Queue(), queue.Queue()]
9
10 def extractWords():
11     url = 'https://timesofindia.indiatimes.com/rssfeeds/1081479906.cms'
12     feed = feedparser.parse(url)
13     for entry in feed['entries'][:5]:
14         text = entry['title']
15         if 'ex' in text:
16             continue
17         words = nltk.word_tokenize(text)
18         data = {'uuid': uuid.uuid4(), 'input': words}
19         queues[0].put(data, True)
20         print(">> {} : {}".format(data['uuid'], text))
21
22 def extractPOS():
23     while True:
24         if queues[0].empty():
25             break
26         else:
27             data = queues[0].get()
28             words = data['input']
29             postags = nltk.pos_tag(words)
30             queues[0].task_done()
31             queues[1].put({'uuid': data['uuid'], 'input': postags}, True)
32
33 def extractNE():
34     while True:
35         if queues[1].empty():
36             break
37         else:
38             data = queues[1].get()
39             postags = data['input']
40             queues[1].task_done()
41             chunks = nltk.ne_chunk(postags, binary=False)
42             print("<< {} : ".format(data['uuid']), end='')
43             for path in chunks:
44                 try:
45                     label = path.label()
46                     print(path, end=', ')
47                 except:
48                     pass
49             print()
50
51 def runProgram():
52     e = threading.Thread(target=extractWords())
53     e.start()
54     threads.append(e)
55
56     p = threading.Thread(target=extractPOS())
57     p.start()
58     threads.append(p)
59
60     n = threading.Thread(target=extractNE())
61     n.start()
62     threads.append(n)
63
64     queues[0].join()
65     queues[1].join()
66
67     for t in threads:
68         t.join()
69
70 if __name__ == '__main__':
71     runProgram()
72

```

3. Save the file.
4. Run the program using the Python interpreter.
5. You will see this output:

```

ch8 — -bash — 103x22
nltk $ python PipelineQ.py
>> 6d07ccce-9bfc-42a7-91f6-41935ae93330 : 'Fukrey Returns' trailer: The 'Jugaadu' boys are back
>> 339476c1-218d-43ba-8d14-a39d949820fd : Akshaye Khanna lands in trouble for smoking on 'Ittefaq' post
ers
>> 70eca8ce-3792-4000-8eff-e9a89cdd684e : Pic: Salman Khan unveils his 'Race 3' look
>> 05f74135-c026-4b60-a720-63b70ef8f21e : Akshay Kumar replaces Salman Khan in 'No Entry' sequel?
>> 23d4949a-1eed-49eb-bec9-6803d6b45315 : Pic: Mira Rajput and daughter Misha Kapoor get smeared in col
ours after a painting class together
<< 6d07ccce-9bfc-42a7-91f6-41935ae93330 : (PERSON Returns/NNP),
<< 339476c1-218d-43ba-8d14-a39d949820fd : (PERSON Akshaye/NNP), (PERSON Khanna/NNP),
<< 70eca8ce-3792-4000-8eff-e9a89cdd684e : (PERSON Salman/NNP Khan/NNP),
<< 05f74135-c026-4b60-a720-63b70ef8f21e : (PERSON Akshay/NNP), (PERSON Kumar/NNP), (PERSON Salman/NNP
Khan/NNP),
<< 23d4949a-1eed-49eb-bec9-6803d6b45315 : (PERSON Mira/NNP Rajput/NNP), (PERSON Misha/NNP Kapoor/NNP)
,
nltk $

```

How it works...

Let's see how to build this pipeline:

```

import nltk
import threading
import queue
import feedparser
import uuid

```

These five instructions import five Python libraries into the current program:

- `nltk`: Natural language toolkit
- `threading`: A threading library used to create lightweight tasks within a single program
- `queue`: A queue library that can be used in a multi-threaded program
- `feedparser`: An RSS feed parsing library
- `uuid`: An RFC-4122-based uuid version 1, 3, 4, 5-generating library

```

threads = []

```

Creating a new empty list to keep track of all the threads in the program:

```
queues = [queue.Queue(), queue.Queue()]
```

This instruction creates a list of two queues in a variable `queue`?

Why do we need two queues:

- The first queue is used to store tokenized sentences
- The second queue is used to store all the POS analyzed words

This instruction defines a new function, `extractWords()`, which reads a sample RSS feed from the internet and stores the words, along with a unique identifier for this text:

```
def extractWords():
```

We are defining a sample URL (entertainment news) from the India Times website:

```
url = 'https://timesofindia.indiatimes.com/rssfeeds/1081479906.cms'
```

This instruction invokes the `parse()` function of the `feedparser` library.

This `parse()` function downloads the content of the URL and converts it into a list of news items. Each news item is a dictionary with title and summary keys:

```
feed = feedparser.parse(url)
```

We are taking the first five entries from the RSS feed and storing the current item in a variable called `entry`:

```
for entry in feed['entries'][:5]:
```

The title of the current RSS feed item is stored in a variable called `text`:

```
text = entry['title']
```

This instruction skips the titles that contain sensitive words. Since we are reading the data from the internet, we have to make sure that the data is properly sanitized:

```
if 'ex' in text:  
    continue
```

Break the input text into words using the `word_tokenize()` function and store the result into a variable called `words`:

```
words = nltk.word_tokenize(text)
```

Create a dictionary called `data` with two key-value pairs, where we are storing the UUID and input words under the `UUID` and `input` keys respectively:

```
data = {'uuid': uuid.uuid4(), 'input': words}
```

This instruction stores the dictionary in the first queue, `queues[0]`. The second argument is set to `true`, which indicates that if the queue is full, pause the thread:

```
queues[0].put(data, True)
```

A well-designed pipeline understands that it should control the inflow and outflow of the data according to the component's computation capacity. If not, the entire pipeline collapses. This instruction prints the current RSS item that we are processing along with its unique ID:

```
print(">> {} : {}".format(data['uuid'], text))
```

This instruction defines a new function called `extractPOS()`, which reads from the first queue, processes the data, and saves the POS of the words in the second queue:

```
def extractPOS():
```

This is an infinite loop:

```
while True:
```

These instructions check whether the first queue is empty. When the queue is empty, we stop processing:

```
if queues[0].empty():  
    break
```

In order to make this program robust, pass the feedback from the first queue. This is left as an exercise to the reader. This is the `else` part, which indicates there is some data in the first queue:

```
else:
```

Take the first item from the queue (in FIFO order):

```
data = queues[0].get()
```

Identify the parts of speech in the words:

```
words = data['input']  
postags = nltk.pos_tag(words)
```

Update the first queue, mentioning that we are done with processing the item that is just extracted by this thread:

```
queues[0].task_done()
```

Store the POS tagged word list in the second queue so that the next phase in the pipeline will execute things. Here also, we are using `true` for the second parameter, which will make sure that the thread will wait if there is no free space in the queue:

```
queues[1].put({'uuid': data['uuid'], 'input': postags}, True)
```

This instruction defines a new function, `extractNE()`, which reads from the second queue, processes the POS tagged words, and prints the named entities on screen:

```
def extractNE():
```

This is an infinite loop instruction:

```
while True:
```

If the second queue is empty, then we exit the infinite loop:

```
if queues[1].empty():
    break
```

This instruction picks an element from the second queue and stores it in a data variable:

```
else:
    data = queues[1].get()
```

This instruction marks the completion of data processing on the element that was just picked from the second queue:

```
postags = data['input']
queues[1].task_done()
```

This instruction extracts the named entities from the `postags` variable and stores it in a variable called `chunks`:

```
chunks = nltk.ne_chunk(postags, binary=False)

print("  << {} : ".format(data['uuid']), end = '')
for path in chunks:
    try:
        label = path.label()
        print(path, end=', ')
    except:
        pass
```

```
print()
```

These instructions do the following

- Print the UUID from the data dictionary
- Iterate over all chunks that are identified
- We are using a try/except block because not all elements in the tree have the `label()` function (they are tuples when no NE is found)
- Finally, we call a `print()` function, which prints a newline on screen

This instruction defines a new function, `runProgram()`, which does the pipeline setup using threads:

```
def runProgram():
```

These three instructions create a new thread with `extractWords()` as the function, start the thread and add the thread object (`e`) to the list called `threads`:

```
e = threading.Thread(target=extractWords())
e.start()
threads.append(e)
```

These instructions create a new thread with `extractPOS()` as the function, start the thread, and add the thread object (`p`) to the list variable `threads`:

```
p = threading.Thread(target=extractPOS())
p.start()
threads.append(p)
```

These instructions create a new thread using `extractNE()` for the code, start the thread, and add the thread object (`n`) to the list `threads`:

```
n = threading.Thread(target=extractNE())
n.start()
threads.append(n)
```

These two instructions release the resources that are allocated to the queues once all the processing is done:

```
queues[0].join()
queues[1].join()
```


These two instructions iterate over the threads list, store the current thread object in a variable, `t`, call the `join()` function to mark the completion of the thread, and release resources allocated to the thread:

```
for t in threads:
    t.join()
```

This is the section of the code that is invoked when the program is run with the main thread. The `runProgram()` is called, which simulates the entire pipeline:

```
if __name__ == '__main__':
    runProgram()
```

Solving the text similarity problem

The text similarity problem deals with the challenge of finding how close given text documents are. Now, when we say close, there are many dimensions in which we can say they are closer or far:

- Sentiment/emotion dimension
- Sense dimension
- Mere presence of certain words

There are many algorithms available for this; all of them vary in the degree of complexity, the resources needed, and the volume of data we are dealing with.

In this recipe, we will use the TF-IDF algorithm to solve the similarity problem. So first, let's understand the basics:

- **Term frequency (TF):** This technique tries to find the relative importance (or frequency) of the word in a given document

Since we are talking about relative importance, we generally normalize the frequency with respect to the total words that are present in the document to compute the TF value of a word.

- **Inverse document frequency (IDF)** : This technique makes sure that words that are frequently used (a, the, and so on) should be given lower weight when compared to the words that are rarely used.

Since both TF and IDF values are decomposed to numbers (fractions), we will do a multiplication of these two values for each term against every document and build M vectors of N dimensions (where N is the total number of documents and M are the unique words in all the documents).

Once we have these vectors, we need to find the cosine similarity using the following formula on these vectors:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|_2 \|B\|_2} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}, \text{ where } A_i \text{ and } B_i \text{ are components of vector } A \text{ and } B \text{ respectively}$$

Getting ready

You should have Python installed, along with the `nltk` and `scikit` libraries. Having some understanding of mathematics is helpful.

How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `Similarity.py`.

3. Type the following source code:

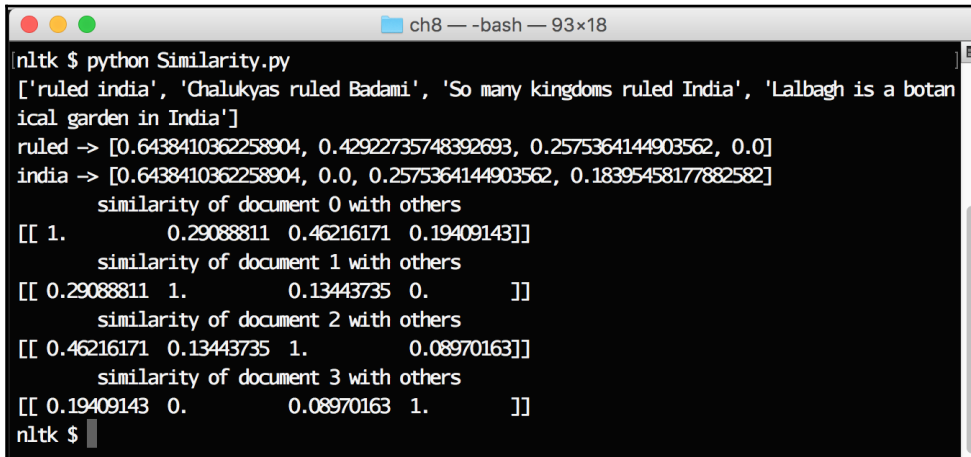
```

Similarity.py

1 import nltk
2 import math
3 from sklearn.feature_extraction.text import TfidfVectorizer
4 from sklearn.metrics.pairwise import cosine_similarity
5
6 class TextSimilarityExample:
7     def __init__(self):
8         self.statements = [
9             'ruled india',
10            'Chalukyas ruled Badami',
11            'So many kingdoms ruled India',
12            'Lalbagh is a botanical garden in India'
13        ]
14    def TF(self, sentence):
15        words = nltk.word_tokenize(sentence.lower())
16        freq = nltk.FreqDist(words)
17        dictionary = {}
18        for key in freq.keys():
19            norm = freq[key]/float(len(words))
20            dictionary[key] = norm
21        return dictionary
22
23    def IDF(self):
24        def idf(TotalNumberOfDocuments, NumberOfDocumentsWithThisWord):
25            return 1.0 +
26                math.log(TotalNumberOfDocuments/NumberOfDocumentsWithThisWord)
27        numDocuments = len(self.statements)
28        uniqueWords = {}
29        idfValues = {}
30        for sentence in self.statements:
31            for word in nltk.word_tokenize(sentence.lower()):
32                if word not in uniqueWords:
33                    uniqueWords[word] = 1
34                else:
35                    uniqueWords[word] += 1
36        for word in uniqueWords:
37            idfValues[word] = idf(numDocuments, uniqueWords[word])
38        return idfValues
39
40    def TF_IDF(self, query):
41        words = nltk.word_tokenize(query.lower())
42        idf = self.IDF()
43        vectors = {}
44        for sentence in self.statements:
45            tf = self.TF(sentence)
46            for word in words:
47                tfv = tf[word] if word in tf else 0.0
48                idfv = idf[word] if word in idf else 0.0
49                mul = tfv * idfv
50                if word not in vectors:
51                    vectors[word] = []
52                vectors[word].append(mul)
53        return vectors
54
55    def displayVectors(self, vectors):
56        print(self.statements)
57        for word in vectors:
58            print("{} -> {}".format(word, vectors[word]))
59
60    def cosineSimilarity(self):
61        vec = TfidfVectorizer()
62        matrix = vec.fit_transform(self.statements)
63        for j in range(1, 5):
64            i = j - 1
65            print("\tsimilarity of document {} with others".format(i))
66            similarity = cosine_similarity(matrix[i:j], matrix)
67            print(similarity)
68
69    def demo(self):
70        inputQuery = self.statements[0]
71        vectors = self.TF_IDF(inputQuery)
72        self.displayVectors(vectors)
73        self.cosineSimilarity()
74
75 similarity = TextSimilarityExample()
76 similarity.demo()

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:



```

ch8 — -bash — 93x18
nltk $ python Similarity.py
['ruled india', 'Chalukyas ruled Badami', 'So many kingdoms ruled India', 'Lalbagh is a botanical garden in India']
ruled -> [0.6438410362258904, 0.42922735748392693, 0.2575364144903562, 0.0]
india -> [0.6438410362258904, 0.0, 0.2575364144903562, 0.18395458177882582]
    similarity of document 0 with others
[[ 1.          0.29088811  0.46216171  0.19409143]]
    similarity of document 1 with others
[[ 0.29088811  1.          0.13443735  0.          ]]
    similarity of document 2 with others
[[ 0.46216171  0.13443735  1.          0.08970163]]
    similarity of document 3 with others
[[ 0.19409143  0.          0.08970163  1.          ]]
nltk $

```

How it works...

Let's see how we are solving the text similarity problem. These four instructions import the necessary libraries that are used in the program:

```

import nltk
import math
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

```

We are defining a new class, `TextSimilarityExample`:

```

class TextSimilarityExample:

```

This instruction defines a new constructor for the class:

```

def __init__(self):

```

This instruction defines sample sentences on which we want to find the similarity.

```
self.statements = [
    'ruled india',
    'Chalukyas ruled Badami',
    'So many kingdoms ruled India',
    'Lalbagh is a botanical garden in India'
]
```

We are defining the TF of all the words in a given sentence:

```
def TF(self, sentence):
    words = nltk.word_tokenize(sentence.lower())
    freq = nltk.FreqDist(words)
    dictionary = {}
    for key in freq.keys():
        norm = freq[key]/float(len(words))
        dictionary[key] = norm
    return dictionary
```

This function does the following things:

- Converts the sentence to lower case and extracts all the words
- Finds the frequency distribution of these words using the `nltk.FreqDist` function
- Iterates over all the dictionary keys, builds the normalized floating values, and stores them in a dictionary
- Returns the dictionary that contains the normalized score for each word in the sentence

We are defining an IDF that finds the IDF value for all the words in all the documents:

```
def IDF(self):
    def idf(TotalNumberOfDocuments, NumberOfDocumentsWithThisWord):
        return 1.0 +
    math.log(TotalNumberOfDocuments/NumberOfDocumentsWithThisWord)
    numDocuments = len(self.statements)
    uniqueWords = {}
    idfValues = {}
    for sentence in self.statements:
        for word in nltk.word_tokenize(sentence.lower()):
            if word not in uniqueWords:
                uniqueWords[word] = 1
            else:
                uniqueWords[word] += 1
    for word in uniqueWords:
        idfValues[word] = idf(numDocuments, uniqueWords[word])
    return idfValues
```

This function does the following things:

- We define a local function called `idf()`, which is the formula to find the IDF of a given word
- We iterate over all the statements and convert them to lowercase
- Find how many times each word is present across all the documents
- Build the IDF value for all words and return the dictionary containing these IDF values

We are now defining a `TF_IDF` (TF multiplied by IDF) for all the documents against a given search string.

```
def TF_IDF(self, query):
    words = nltk.word_tokenize(query.lower())
    idf = self.IDF()
    vectors = {}
    for sentence in self.statements:
        tf = self.TF(sentence)
        for word in words:
            tfv = tf[word] if word in tf else 0.0
            idfv = idf[word] if word in idf else 0.0
            mul = tfv * idfv
            if word not in vectors:
                vectors[word] = []
            vectors[word].append(mul)
    return vectors
```

Let's see what we are doing here:

- Break the search string into tokens
- Build `IDF()` for all sentences in the `self.statements` variable
- Iterate over all sentences and find the TF for all words in this sentence
- Filter and use only the words that are present in the input search string and build vectors that consist of $tf \times idf$ values against each document
- Return the list of vectors for each word in the search query

This function displays the contents of vectors on screen:

```
def displayVectors(self, vectors):
    print(self.statements)
    for word in vectors:
        print("{} -> {}".format(word, vectors[word]))
```

Now, in order to find the similarity, as we discussed initially, we need to find the Cosine similarity on all the input vectors. We can do all the math ourselves. But this time, let's try to use scikit to do all the computations for us.

```
def cosineSimilarity(self):
    vec = TfidfVectorizer()
    matrix = vec.fit_transform(self.statements)
    for j in range(1, 5):
        i = j - 1
        print("\tsimilarity of document {} with others".format(i))
        similarity = cosine_similarity(matrix[i:], matrix)
        print(similarity)
```

In the previous functions, we learned how to build TF and IDF values and finally get the TF x IDF values for all the documents.

Let's see what we are doing here:

- Defining a new function: `cosineSimilarity()`
- Creating a new vectorizer object
- Building a matrix of TF-IDF values for all the documents that we are interested in, using the `fit_transform()` function
- Later we compare each document with all other documents and see how close they are to each other

This is the `demo()` function and it runs all the other functions we have defined before:

```
def demo(self):
    inputQuery = self.statements[0]
    vectors = self.TF_IDF(inputQuery)
    self.displayVectors(vectors)
    self.cosineSimilarity()
```

Let's see what we are doing here

- We take the first statement as our input query.
- We build vectors using our own handwritten `TF_IDF()` function.
- We display our TF x IDF vectors for all sentences on screen.
- We print the cosine similarity computed for all the sentences using the `scikit` library by invoking the `cosineSimilarity()` function.

We are creating a new object for the `TextSimilarityExample()` class and then invoking the `demo()` function.

```
similarity = TextSimilarityExample()  
similarity.demo()
```

Identifying topics

In the previous chapter, we learned how to do document classification. Beginners might think document classification and topic identification are the same, but there is a slight difference.

Topic identification is the process of discovering topics that are present in the input document set. These topics can be multiple words that occur uniquely in a given text.

Let's take an example. When we read arbitrary text that contains a mention of Sachin Tendulkar, score, win we can understand that the sentence is describing cricket. But we may be wrong as well.

In order to find all these types of topics in a given input text, we use the Latent Dirichlet allocation algorithm (we could use TF-IDF as well, but since we have already explored it in a previous recipe, let's see how LDA works in identifying the topic).

Getting ready

You should have Python installed, along with the `nltk`, `gensim`, and `feedparser` libraries.

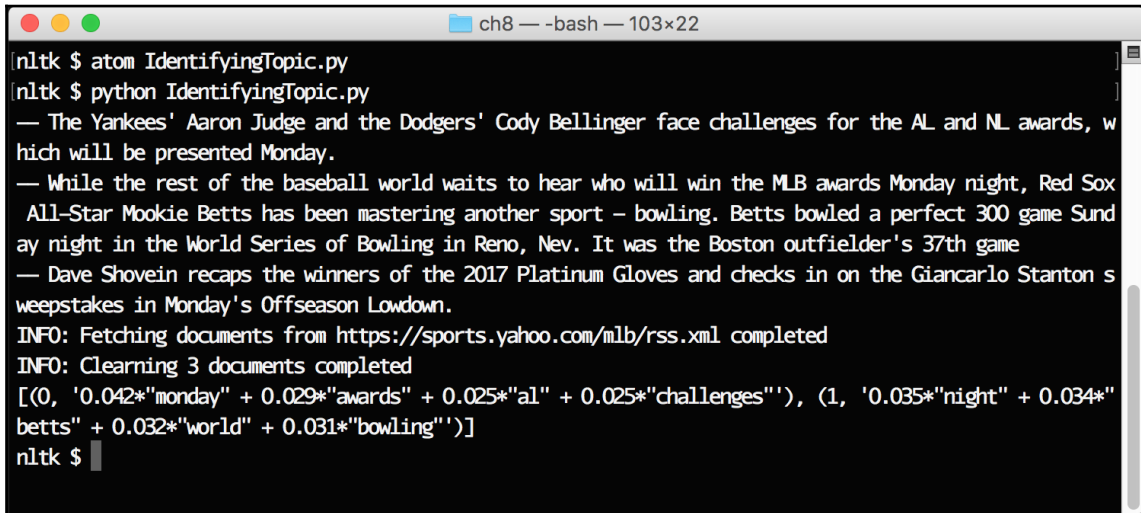
How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `IdentifyingTopic.py`.
3. Type the following source code:

IdentifyingTopic.py

```
1 from nltk.tokenize import RegexpTokenizer
2 from nltk.corpus import stopwords
3 from gensim import corpora, models
4 import nltk
5 import feedparser
6
7 class IdentifyingTopicExample:
8     def getDocuments(self):
9         url = 'https://sports.yahoo.com/mlb/rss.xml'
10        feed = feedparser.parse(url)
11        self.documents = []
12        for entry in feed['entries'][:5]:
13            text = entry['summary']
14            if 'ex' in text:
15                continue
16            self.documents.append(text)
17            print("-- {}".format(text))
18        print("INFO: Fetching documents from {} completed".format(url))
19
20    def cleanDocuments(self):
21        tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
22        en_stop = set(stopwords.words('english'))
23        self.cleaned = []
24        for doc in self.documents:
25            lowercase_doc = doc.lower()
26            words = tokenizer.tokenize(lowercase_doc)
27            non_stopped_words = [i for i in words if not i in en_stop]
28            self.cleaned.append(non_stopped_words)
29        print("INFO: Clearing {} documents
30        * completed".format(len(self.documents)))
31
32    def doLDA(self):
33        dictionary = corpora.Dictionary(self.cleaned)
34        corpus = [dictionary.doc2bow(cleandoc) for cleandoc in self.cleaned]
35        * ldamodel = models.ldamodel.LdaModel(corpus, num_topics=2, id2word =
36        dictionary)
37        print(ldamodel.print_topics(num_topics=2, num_words=4))
38
39    def run(self):
40        self.getDocuments()
41        self.cleanDocuments()
42        self.doLDA()
43
44 if __name__ == '__main__':
45     topicExample = IdentifyingTopicExample()
46     topicExample.run()
```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:

A terminal window titled 'ch8 — -bash — 103x22' showing the execution of a Python script. The user enters 'nlTK \$ atom IdentifyingTopic.py' and 'nlTK \$ python IdentifyingTopic.py'. The script outputs a list of sentences about baseball awards and bowling, followed by status messages 'INFO: Fetching documents from https://sports.yahoo.com/mlb/rss.xml completed' and 'INFO: Cleaning 3 documents completed'. Finally, it prints a list of topic distributions for each document, such as '(0, '0.042*monday' + 0.029*awards' + 0.025*al' + 0.025*challenges')' and '(1, '0.035*night' + 0.034*betts' + 0.032*world' + 0.031*bowling')'. The prompt 'nlTK \$' is visible at the bottom.

```
nlTK $ atom IdentifyingTopic.py
nlTK $ python IdentifyingTopic.py
— The Yankees' Aaron Judge and the Dodgers' Cody Bellinger face challenges for the AL and NL awards, which will be presented Monday.
— While the rest of the baseball world waits to hear who will win the MLB awards Monday night, Red Sox All-Star Mookie Betts has been mastering another sport – bowling. Betts bowled a perfect 300 game Sunday night in the World Series of Bowling in Reno, Nev. It was the Boston outfielder's 37th game
— Dave Shovein recaps the winners of the 2017 Platinum Gloves and checks in on the Giancarlo Stanton's weepstakes in Monday's Offseason Lowdown.
INFO: Fetching documents from https://sports.yahoo.com/mlb/rss.xml completed
INFO: Cleaning 3 documents completed
[(0, '0.042*monday' + 0.029*awards' + 0.025*al' + 0.025*challenges'), (1, '0.035*night' + 0.034*betts' + 0.032*world' + 0.031*bowling')]
nlTK $
```

How it works...

Let's see how the topic identification program works. These five instructions import the necessary libraries into the current program.

```
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from gensim import corpora, models
import nltk
import feedparser
```

This instruction defines a new class, `IdentifyingTopicExample`:

```
class IdentifyingTopicExample:
```

This instruction defines a new function, `getDocuments()`, whose responsibility is to download few documents from the internet using `feedparser`:

```
def getDocuments(self):
```

Download all the documents mentioned in the URL and store the list of dictionaries into a variable called `feed`:

```
url = 'https://sports.yahoo.com/mlb/rss.xml'
feed = feedparser.parse(url)
```

Empty the list to keep track of all the documents that we are going to analyze further:

```
self.documents = []
```

Take the top five documents from the `feed` variable and store the current news item into a variable called `entry`:

```
for entry in feed['entries'][:5]:
```

Store the news summary into a variable called `text`:

```
text = entry['summary']
```

If the news article contains any sensitive words, skip those:

```
if 'ex' in text:
    continue
```

Store the document in the `documents` variable:

```
self.documents.append(text)
```

Display the current document on the screen:

```
print("-- {}".format(text))
```

Display an informational message to the user that we have collected N documents from the given `url`:

```
print("INFO: Fetching documents from {} completed".format(url))
```

This instruction defines a new function, `cleanDocuments()`, whose responsibility is to clean the input text (since we are downloading it from the internet, it can contain any type of data).

```
def cleanDocuments(self):
```

We are interested in extracting words that are in the English alphabet. So, this tokenizer is defined to break the text into tokens, where each token consists of letters from a to z and A-Z. By doing so, we can be sure that punctuation and other bad data doesn't come into the processing.

```
tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
```

Store the stop words of English in a variable, `en_stop`:

```
en_stop = set(stopwords.words('english'))
```

Define a empty list called `cleaned`, which is used to store all the cleaned and tokenized documents:

```
self.cleaned = []
```

Iterate over all the documents we have collected using the `getDocuments()` function:

```
for doc in self.documents:
```

Convert the document to lowercase to avoid treating the same word differently because they are case sensitive:

```
lowercase_doc = doc.lower()
```

Break the sentence into words. The output is a list of words stored in a variable called `words`:

```
words = tokenizer.tokenize(lowercase_doc)
```

Ignore all the words from the sentence if they belong to the English stop word category and store all of them in the `non_stopped_words` variable:

```
non_stopped_words = [i for i in words if not i in en_stop]
```

Store the sentence that is tokenized and cleaned in a variable called `self.cleaned` (class member).

```
self.cleaned.append(non_stopped_words)
```

Show a diagnostic message to the user that we have finished cleaning the documents:

```
print("INFO: Cleaning {} documents  
completed".format(len(self.documents)))
```

This instruction defines a new function, `doLDA`, which runs the LDA analysis on the cleaned documents:

```
def doLDA(self):
```

Before we directly process the cleaned documents, we create a dictionary from these documents:

```
dictionary = corpora.Dictionary(self.cleaned)
```

The input corpus is defined as a bag of words for each cleaned sentence:

```
corpus = [dictionary.doc2bow(cleandoc) for cleandoc in self.cleaned]
```

Create a model on the corpus with the number of topics defined as 2 and set the vocabulary size/mapping using the `id2word` parameter:

```
ldamodel = models.ldamodel.LdaModel(corpus, num_topics=2, id2word =  
dictionary)
```

Print two topics, where each topic should contain four words on the screen:

```
print(ldamodel.print_topics(num_topics=2, num_words=4))
```

This is the function that does all the steps in order:

```
def run(self):  
    self.getDocuments()  
    self.cleanDocuments()  
    self.doLDA()
```

When the current program is invoked as the main program, create a new object called `topicExample` from the `IdentifyingTopicExample()` class and invoke the `run()` function on the object.

```
if __name__ == '__main__':  
    topicExample = IdentifyingTopicExample()  
    topicExample.run()
```

Summarizing text

In this information overload era, there is so much information that is available in print/text form. Its humanly impossible for us to consume all this data. In order to make the consumption of this data easier, we have been trying to invent algorithms that can help simplify large text into a summary (or a gist) that we can easily digest.

By doing this, we will save time and also make things easier for the network.

In this recipe, we will use the `gensim` library, which has built-in support for this summarization using the `TextRank` algorithm (<https://web.eecs.umich.edu/~mihalcea/papers/mihalcea.emnlp04.pdf>).

Getting ready

You should have Python installed, along with the `bs4` and `gensim` libraries.

How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `Summarize.py`.

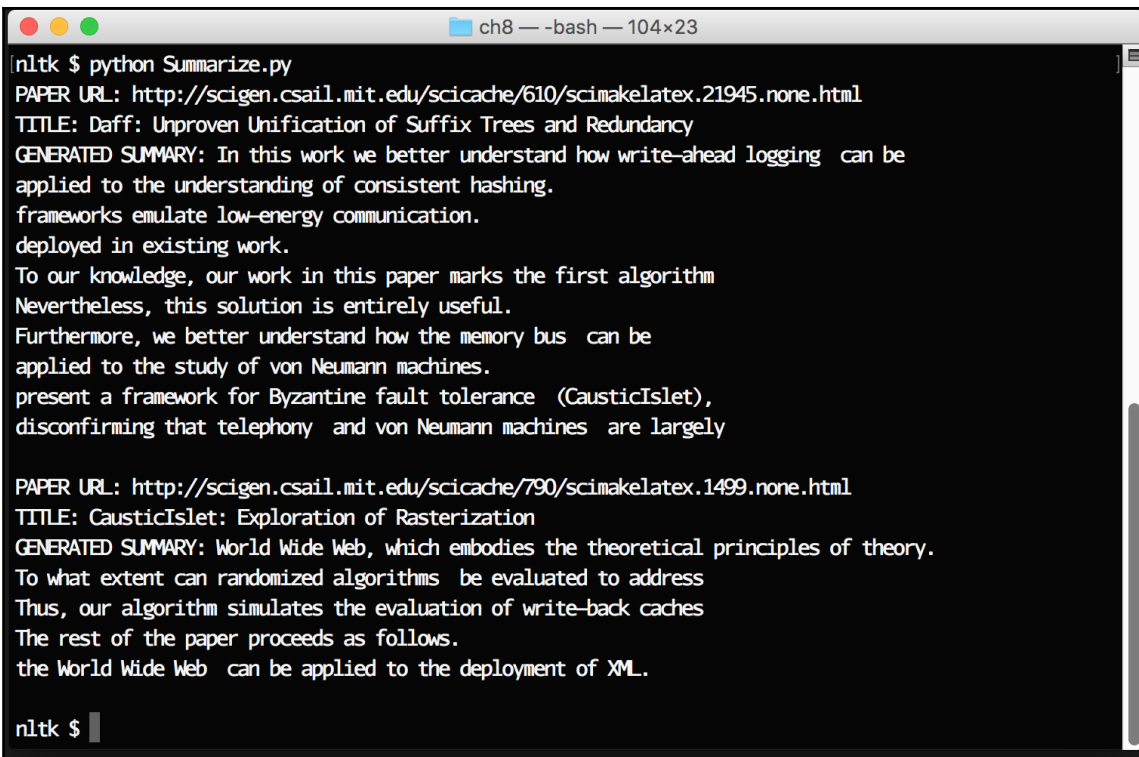
3. Type the following source code:

```
Summarize.py
1 from gensim.summarization import summarize
2 from bs4 import BeautifulSoup
3 import requests
4
5 #
6 # This recipe uses automatic computer science Paper generation tool from mit.edu
7 # You can generate your own paper by visiting
8 # https://pdos.csail.mit.edu/archive/scigen/
9 # and click generate.
10 #
11 # This example needs large amount of text that needs to be available for
12 # summary.
13 # So, we are using this paper generation tool and extracting the 'Introduction'
14 # section
15 # to do the summary analysis.
16 #
17 urls = {
18     'Daff: Unproven Unification of Suffix Trees and Redundancy':
19     * http://scigen.csail.mit.edu/scicache/610/scimakelateX.21945.none.html,
20     'CausticIslet: Exploration of Rasterization':
21     * http://scigen.csail.mit.edu/scicache/790/scimakelateX.1499.none.html
22 }
23
24 for key in urls.keys():
25     url = urls[key]
26     r = requests.get(url)
27     soup = BeautifulSoup(r.text, 'html.parser')
28     data = soup.get_text()
29     pos1 = data.find("1 Introduction") + len("1 Introduction")
30     pos2 = data.find("2 Related Work")
31     text = data[pos1:pos2].strip()
32     print("PAPER URL: {}".format(url))
33     print("TITLE: {}".format(key))
34     print("GENERATED SUMMARY: {}".format(summarize(text)))
35     print()
```

4. Save the file.

5. Run the program using the Python interpreter.

6. You will see the following output:

A terminal window titled 'ch8 — -bash — 104x23' displays the output of a Python script. The script uses NLTK to fetch and summarize two papers. The first paper is 'Daff: Unproven Unification of Suffix Trees and Redundancy' and the second is 'CausticIslet: Exploration of Rasterization'. The output shows the paper URLs, titles, and generated summaries for each.

```
nltk $ python Summarize.py
PAPER URL: http://scigen.csail.mit.edu/scicache/610/scimake latex.21945.none.html
TITLE: Daff: Unproven Unification of Suffix Trees and Redundancy
GENERATED SUMMARY: In this work we better understand how write-ahead logging can be
applied to the understanding of consistent hashing.
frameworks emulate low-energy communication.
deployed in existing work.
To our knowledge, our work in this paper marks the first algorithm
Nevertheless, this solution is entirely useful.
Furthermore, we better understand how the memory bus can be
applied to the study of von Neumann machines.
present a framework for Byzantine fault tolerance (CausticIslet),
disconfirming that telephony and von Neumann machines are largely

PAPER URL: http://scigen.csail.mit.edu/scicache/790/scimake latex.1499.none.html
TITLE: CausticIslet: Exploration of Rasterization
GENERATED SUMMARY: World Wide Web, which embodies the theoretical principles of theory.
To what extent can randomized algorithms be evaluated to address
Thus, our algorithm simulates the evaluation of write-back caches
The rest of the paper proceeds as follows.
the World Wide Web can be applied to the deployment of XML.

nltk $
```

How it works...

Let's see how our summarization program works.

```
from gensim.summarization import summarize
from bs4 import BeautifulSoup
import requests
```

These three instructions import the necessary libraries into the current program:

- `gensim.summarization.summarize`: Text-rank-based summarization algorithm
- `bs4`: A BeautifulSoup library for parsing HTML documents
- `requests`: A library to download HTTP resources

We are defining a dictionary called `urls` whose keys are the title of the paper that is auto-generated and the value is the URL to the paper:

```
urls = {
    'Daff: Unproven Unification of Suffix Trees and Redundancy':
    'http://scigen.csail.mit.edu/scicache/610/scimakelatem.21945.none.html',
    'CausticIslet: Exploration of Rasterization':
    'http://scigen.csail.mit.edu/scicache/790/scimakelatem.1499.none.html'
}
```

Iterate through all the keys of the dictionary:

```
for key in urls.keys():
```

Store the URL of the current paper in a variable called `url`:

```
url = urls[key]
```

Download the content of the url using the `requests` library's `get()` method and store the response object into a variable, `r`:

```
r = requests.get(url)
```

Use `BeautifulSoup()` to parse the text from the `r` object using the HTML parser and store the return object in a variable called `soup`:

```
soup = BeautifulSoup(r.text, 'html.parser')
```

Strip out all the HTML tags and extract only the text from the document into the variable `data`:

```
data = soup.get_text()
```

Find the position of the text `Introduction` and skip past towards end of string, to mark is as starting offset from which we want to extract the substring.

```
pos1 = data.find("1 Introduction") + len("1 Introduction")
```

Find the second position in the document, exactly at the beginning of the related work section:

```
pos2 = data.find("2 Related Work")
```

Now, extract the introduction of the paper, which is between these two offsets:

```
text = data[pos1:pos2].strip()
```

Display the URL and the title of the paper on the screen:

```
print("PAPER URL: {}".format(url))
print("TITLE: {}".format(key))
```

Call the `summarize()` function on the text, which returns shortened text as per the text rank algorithm:

```
print("GENERATED SUMMARY: {}".format(summarize(text)))
```

Print an extra newline for more readability of the screen output.

```
print()
```

Resolving anaphora

In many natural languages, while forming sentences, we avoid the repeated use of certain nouns with pronouns to simplify the sentence construction.

For example:

Ravi is a boy. He often donates money to the poor.

In this example, there are two statements:

- Ravi is a boy.
- He often donates money to the poor.

When we start analyzing the second statement, we cannot make a decision about who is donating the money without knowing about the first statement. So, we should associate He with Ravi to get the complete sentence meaning. All this reference resolution happens naturally in our mind.

If we observe the previous example carefully, first the subject is present; then the pronoun comes up. So the direction of the flow is from left to right. Based on this flow, we can call these types of sentences anaphora.

Let's take another example:

He was already on his way to airport. Realized Ravi

This is another class of example where the direction of expression is the reverse order (first the pronoun and then the noun). Here too, He is associated with Ravi. These types of sentences are called as Cataphora.

The earliest available algorithm for this anaphora resolution dates back to the 1970; Hobbs has presented a paper on this. An online version of this paper is available here: <https://www.isi.edu/~hobbs/pronoun-papers.html>.

In this recipe, we will try to write a very simple Anaphora resolution algorithm using what we have learned just now.

Getting ready

You should have python installed, along with the `nltk` library and `gender` datasets.

You can use `nltk.download()` to download the corpus.

How to do it...

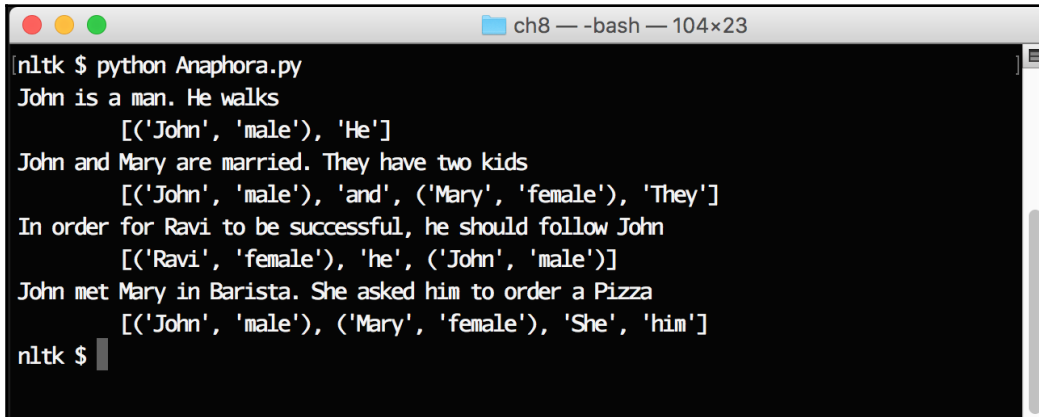
1. Open atom editor (or your favorite programming editor).
2. Create a new file called `Anaphora.py`.

3. Type the following source code:

```
Anaphora.py
1 import nltk
2 from nltk.chunk import tree2conlltags
3 from nltk.corpus import names
4 import random
5
6 class AnaphoraExample:
7     def __init__(self):
8         males = [(name, 'male') for name in names.words('male.txt')]
9         females = [(name, 'female') for name in names.words('female.txt')]
10        combined = males + females
11        random.shuffle(combined)
12        training = [(self.feature(name), gender) for (name, gender) in combined]
13        self._classifier = nltk.NaiveBayesClassifier.train(training)
14
15    def feature(self, word):
16        return {'last(1)' : word[-1]}
17
18    def gender(self, word):
19        return self._classifier.classify(self.feature(word))
20
21    def learnAnaphora(self):
22        sentences = [
23            "John is a man. He walks",
24            "John and Mary are married. They have two kids",
25            "In order for Ravi to be successful, he should follow John",
26            "John met Mary in Barista. She asked him to order a Pizza"
27        ]
28
29        for sent in sentences:
30            chunks = nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(sent)),
31                                   binary=False)
32            stack = []
33            print(sent)
34            items = tree2conlltags(chunks)
35            for item in items:
36                if item[1] == 'NNP' and (item[2] == 'B-PERSON' or item[2] ==
37                                       'O'):
38                    stack.append((item[0], self.gender(item[0])))
39                elif item[1] == 'CC':
40                    stack.append(item[0])
41                elif item[1] == 'PRP':
42                    stack.append(item[0])
43            print("\t {}".format(stack))
44
45    anaphora = AnaphoraExample()
46    anaphora.learnAnaphora()
```

4. Save the file.

5. Run the program using the Python interpreter.
6. You will see the following output:

A terminal window titled 'ch8 — -bash — 104x23' showing the execution of a Python script. The prompt is 'nltk \$'. The script outputs five sentences followed by their corresponding anaphoric references in tuple format. The references are: 1. [('John', 'male'), 'He'], 2. [('John', 'male'), 'and', ('Mary', 'female'), 'They'], 3. [('Ravi', 'female'), 'he', ('John', 'male')], 4. [('John', 'male'), ('Mary', 'female'), 'She', 'him'], and 5. The prompt returns to 'nltk \$'.

How it works...

Let see how our simple Anaphora resolution algorithm works.

```
import nltk
from nltk.chunk import tree2conlltags
from nltk.corpus import names
import random
```

These four instructions import the necessary modules and functions that are used in the program. We are defining a new class called `AnaphoraExample`:

```
class AnaphoraExample:
```

We are defining a new constructor for this class, which doesn't take any parameters:

```
def __init__(self):
```

These two instructions load all the male and female names from the `nltk.names` corpus and tag them as male/female before storing them in two lists called `male/female`.

```
males = [(name, 'male') for name in names.words('male.txt')]
females = [(name, 'female') for name in names.words('female.txt')]
```

This instruction creates a unique list of males and females. `random.shuffle()` ensures that all of the data in the list is randomized:

```
combined = males + females
random.shuffle(combined)
```

This instruction invokes the `feature()` function on the gender and stores all the names in a variable called `training`:

```
training = [(self.feature(name), gender) for (name, gender) in
combined]
```

We are creating a `NaiveBayesClassifier` object called `_classifier` using the males and females features that are stored in a variable called `training`:

```
self._classifier = nltk.NaiveBayesClassifier.train(training)
```

This function defines the simplest possible feature, which categorizes the given name as male or female just by looking at the last letter of the name:

```
def feature(self, word):
    return {'last(1)' : word[-1]}
```

This function takes a word as an argument and tries to detect the gender as male or female using the classifier we have built:

```
def gender(self, word):
    return self._classifier.classify(self.feature(word))
```

This is the main function that is of interest to us, as we are going to detect anaphora on the sample sentences:

```
def learnAnaphora(self):
```

These are four examples with mixed complexity expressed in anaphora form:

```
sentences = [
    "John is a man. He walks",
    "John and Mary are married. They have two kids",
    "In order for Ravi to be successful, he should follow John",
    "John met Mary in Barista. She asked him to order a Pizza"
]
```

This instruction iterates over all the sentences by taking one sentence at a time to a local variable called `sent`:

```
for sent in sentences:
```

This instruction tokenizes, assigns parts of speech, extracts chunks (named entities), and returns the chunk tree to a variable called `chunks`:

```
chunks = nltk.ne_chunk(nltk.pos_tag(nltk.word_tokenize(sent)),
    binary=False)
```

This variable is used to store all the names and pronouns that help us resolve anaphora:

```
stack = []
```

This instruction shows the current sentence that is being processed on the user's screen:

```
print(sent)
```

This instruction flattens the tree chunks to a list of items expressed in IOB format:

```
items = tree2conlltags(chunks)
```

We are traversing through all chunked sentences that are in IOB format (tuple with three elements):

```
for item in items:
```

If the POS of the word is NNP and IOB letter for this word is B-PERSON or O, then we mark this word as a Name:

```
    if item[1] == 'NNP' and (item[2] == 'B-PERSON' or item[2]
        == 'O'):
        stack.append((item[0], self.gender(item[0])))
```

If the POS of the word is CC, then also we will add this to the `stack` variable:

```
    elif item[1] == 'CC':
        stack.append(item[0])
```

If the POS of the word is PRP, then we will add this to the `stack` variable:

```
    elif item[1] == 'PRP':
        stack.append(item[0])
```

Finally we print the stack on the screen:

```
print("\t {}".format(stack))
```

We are creating a new object called `anaphora` from `AnaphoraExample()` and invoking the `learnAnaphora()` function on the `anaphora` object. Once this function execution completes, we see the list of words for every sentence.

```
anaphora = AnaphoraExample()  
anaphora.learnAnaphora()
```

Disambiguating word sense

In previous chapters, we learned how to identify POS of the words, find named entities, and so on. Just like a word in English behaves as both a noun and a verb, finding the sense in which a word is used is very difficult for computer programs.

Let's take a few examples to understand this sense portion:

Sentence	Description
<i>She is my date</i>	Here the sense of the word <i>date</i> is not the calendar date but expresses a human relationship.
<i>You have taken too many leaves to skip cleaning leaves in the garden</i>	Here the word <i>leaves</i> has multiple senses: <ul style="list-style-type: none">• The first word <i>leave</i> means taking a break• The second one actually refers to tree leaves

Like this, there are many combinations of senses possible in sentences.

One of the challenges we have faced for senses identification is to find a proper nomenclature to describe these senses. There are many English dictionaries available that describe the behavior of words and all possible combinations of those. Of them all, WordNet is the most structured, preferred, and widely accepted source of sense usage.

In this recipe, we will see examples of senses from the WordNet library and use the built-in `nltk` library to find out the sense of words.

Lesk is the oldest algorithm that was coined to tackle this sense detection. You will see, however, that this one too is not accurate in some cases.

Getting ready

You should have Python installed, along with the `nltk` library.

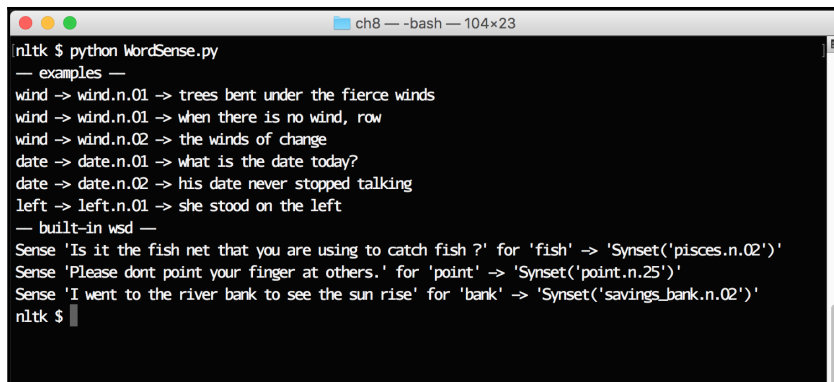
How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `WordSense.py`.
3. Type the following source code:

```
WordSense.py
1  import nltk
2
3  def understandWordSenseExamples():
4      words = ['wind', 'date', 'left']
5      print("-- examples --")
6      for word in words:
7          syns = nltk.corpus.wordnet.synsets(word)
8          for syn in syns[:2]:
9              for example in syn.examples()[:2]:
10                 print("{} -> {} -> {}".format(word, syn.name(), example))
11
12
13 def understandBuiltinWSD():
14     print("-- built-in wsd --")
15     maps = [
16         ('Is it the fish net that you are using to catch fish?', 'fish', 'n'),
17         ('Please dont point your finger at others.', 'point', 'n'),
18         ('I went to the river bank to see the sun rise', 'bank', 'n'),
19     ]
20     for m in maps:
21         print("Sense '{}' for '{}' -> {}".format(m[0], m[1],
22             * nltk.wsd.lesk(m[0], m[1], m[2])))
23
24 if __name__ == '__main__':
25     understandWordSenseExamples()
26     understandBuiltinWSD()
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:



```

nltk $ python WordSense.py
-- examples --
wind -> wind.n.01 -> trees bent under the fierce winds
wind -> wind.n.01 -> when there is no wind, row
wind -> wind.n.02 -> the winds of change
date -> date.n.01 -> what is the date today?
date -> date.n.02 -> his date never stopped talking
left -> left.n.01 -> she stood on the left
-- built-in wsd --
Sense 'Is it the fish net that you are using to catch fish?' for 'fish' -> 'Synset('piscis.n.02')'
Sense 'Please dont point your finger at others.' for 'point' -> 'Synset('point.n.25')'
Sense 'I went to the river bank to see the sun rise' for 'bank' -> 'Synset('savings_bank.n.02')'
nltk $

```

How it works...

Let's see how our program works. This instruction imports the `nltk` library into the program:

```
import nltk
```

We are defining a function with the name `understandWordSenseExamples()`, which uses the WordNet corpus to showcase the possible senses of the words that we are interested in.

```
def understandWordSenseExamples():
```

These are the three words with different senses of expression. They are stored as a list in a variable called `words`.

```

words = ['wind', 'date', 'left']

print("-- examples --")
for word in words:
    syns = nltk.corpus.wordnet.synsets(word)
    for syn in syns[:2]:
        for example in syn.examples()[:2]:
            print("{} -> {} -> {}".format(word, syn.name(), example))

```

These instructions do the following:

- Iterate over all the words in the list by storing the current word in a variable called `word`.

- Invoke the `synsets()` function from the `wordnet` module and store the result in the `syns` variable.
- Take the first three synsets from the list, iterate through them, and take the current one in a variable called `syn`.
- Invoke the `examples()` function on the `syn` object and take the first two examples as the iterator. The current value of the iterator is available in the variable `example`.
- Print the word, synset's name, and example sentence finally.

Define a new function, `understandBuiltinWSD()`, to explore the NLTK built-in `lesk` algorithm's performance on sample sentences.

```
def understandBuiltinWSD():
```

Define a new variable called `maps`, a list of tuples.

```
    print("-- built-in wsd --")
    maps = [
        ('Is it the fish net that you are using to catch fish?', 'fish',
         'n'),
        ('Please dont point your finger at others.', 'point', 'n'),
        ('I went to the river bank to see the sun rise', 'bank', 'n'),
    ]
```

Each tuple consists of three elements:

- The sentence we want to analyze
- The word in the sentence for which we want to find the sense
- The POS of the word

In these two instructions, we are traversing through the `maps` variable, taking the current tuple into variable `m`, invoking the `nltk.wsd.lesk()` function, and displaying the formatted results on screen.

```
    for m in maps:
        print("Sense '{}' for '{}' -> {}".format(m[0], m[1],
          nltk.wsd.lesk(m[0], m[1], m[2])))
```

When the program is run, call the two functions that show the results on the user's screen.

```
if __name__ == '__main__':
    understandWordSenseExamples()
    understandBuiltinWSD()
```

Performing sentiment analysis

Feedback is one of the most powerful measures for understanding relationships. Humans are very good at understanding feedback in verbal communication as the analysis happens unconsciously. In order to write computer programs that can measure and find the emotional quotient, we should have some good understanding of the ways these emotions are expressed in these natural languages.

Let's take a few examples:

Sentence	Description
<i>I am very happy</i>	Indicates a happy emotion
<i>She is so :(</i>	We know there is an iconic sadness expression here

With the increased use of text, icons, and emojis in written natural language communication, it's becoming increasingly difficult for computer programs to understand the emotional meaning of a sentence.

Let's try to write a program to understand the facilities `nltk` provides to build our own algorithm.

Getting ready

You should have Python installed, along with the `nltk` library.

How to do it...

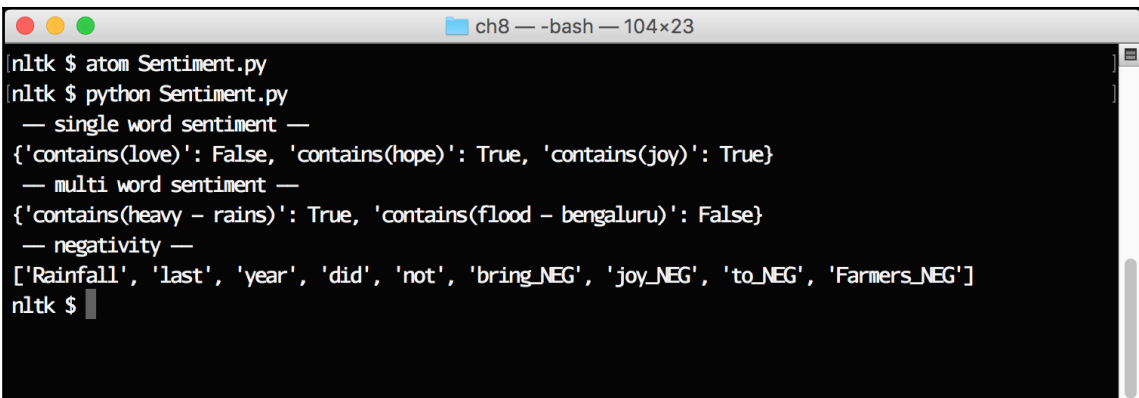
1. Open atom editor (or your favorite programming editor).
2. Create a new file called `Sentiment.py`.

3. Type the following source code:

```
Sentiment.py
1 import nltk
2 import nltk.sentiment.sentiment_analyzer
3
4 def wordBasedSentiment():
5     positive_words = ['love', 'hope', 'joy']
6     text = 'Rainfall this year brings lot of hope and joy to Farmers.'.split()
7     analysis = nltk.sentiment.util.extract_unigram_feats(text, positive_words)
8     print(' -- single word sentiment --')
9     print(analysis)
10
11 def multiWordBasedSentiment():
12     word_sets = [('heavy', 'rains'), ('flood', 'bengaluru')]
13     text = 'heavy rains cause flash flooding in bengaluru'.split()
14     analysis = nltk.sentiment.util.extract_bigram_feats(text, word_sets)
15     print(' -- multi word sentiment --')
16     print(analysis)
17
18 def markNegativity():
19     text = 'Rainfall last year did not bring joy to Farmers'.split()
20     negation = nltk.sentiment.util.mark_negation(text)
21     print(' -- negativity --')
22     print(negation)
23
24 if __name__ == '__main__':
25     wordBasedSentiment()
26     multiWordBasedSentiment()
27     markNegativity()
28
```

4. Save the file.
5. Run the program using the Python interpreter.

6. You will see the following output:

A terminal window titled 'ch8 — -bash — 104x23' showing the execution of a script. The user enters 'nltk \$ atom Sentiment.py' and 'nltk \$ python Sentiment.py'. The script outputs: '— single word sentiment —', a dictionary {'contains(love)': False, 'contains(hope)': True, 'contains(joy)': True}, '— multi word sentiment —', a dictionary {'contains(heavy - rains)': True, 'contains(flood - bengaluru)': False}, '— negativity —', and a list ['Rainfall', 'last', 'year', 'did', 'not', 'bring_NEG', 'joy_NEG', 'to_NEG', 'Farmers_NEG']. The prompt 'nltk \$' is shown at the bottom.

```
nltk $ atom Sentiment.py
nltk $ python Sentiment.py
— single word sentiment —
{'contains(love)': False, 'contains(hope)': True, 'contains(joy)': True}
— multi word sentiment —
{'contains(heavy - rains)': True, 'contains(flood - bengaluru)': False}
— negativity —
['Rainfall', 'last', 'year', 'did', 'not', 'bring_NEG', 'joy_NEG', 'to_NEG', 'Farmers_NEG']
nltk $
```

How it works...

Let's see how our sentiment analysis program works. These instructions import the `nltk` module and `sentiment_analyzer` module respectively.

```
import nltk
import nltk.sentiment.sentiment_analyzer
```

We are defining a new function, `wordBasedSentiment()`, which we will use to learn how to do sentiment analysis based on the words that we already know and which mean something important to us.

```
def wordBasedSentiment():
```

We are defining a list of three words that are special to us as they represent some form of happiness. These words are stored in the `positive_words` variable.

```
positive_words = ['love', 'hope', 'joy']
```

This is the sample text that we are going to analyze; the text is stored in a variable called `text`.

```
text = 'Rainfall this year brings lot of hope and joy to
Farmers.'.split()
```

We are calling the `extract_unigram_feats()` function on the text using the words that we have defined. The result is a dictionary of input words that indicate whether the given words are present in the text or not.

```
analysis = nltk.sentiment.util.extract_unigram_feats(text,
positive_words)
```

This instruction displays the dictionary on the user's screen.

```
print(' -- single word sentiment --')
print(analysis)
```

This instruction defines a new function that we will use to understand whether some pairs of words occur in a sentence.

```
def multiWordBasedSentiment():
```

This instruction defines a list of two-word tuples. We are interested in finding if these pairs of words occur together in a sentence.

```
word_sets = [('heavy', 'rains'), ('flood', 'bengaluru')]
```

This is the sentence we are interested in processing and finding the features of.

```
text = 'heavy rains cause flash flooding in bengaluru'.split()
```

We are calling the `extract_bigram_feats()` on the input sentence against the sets of words in the `word_sets` variable. The result is a dictionary that tells whether these pairs of words are present in the sentence or not.

```
analysis = nltk.sentiment.util.extract_bigram_feats(text, word_sets)
```

This instruction displays the dictionary on screen.

```
print(' -- multi word sentiment --')
print(analysis)
```

We are defining a new function, `markNegativity()`, which helps us understand how we can find negativity in a sentence.

```
def markNegativity():
```

Next is the sentence on which we want to run the negativity analysis. It's stored in a variable, `text`.

```
text = 'Rainfall last year did not bring joy to Farmers'.split()
```

We are calling the `mark_negation()` function on the text. This returns a list of all the words in the sentence along with a special suffix `_NEG` for all the words that come under the negative sense. The result is stored in the `negation` variable.

```
negation = nltk.sentiment.util.mark_negation(text)
```

This instruction displays the list `negation` on screen.

```
print(' -- negativity --')
print(negation)
```

When the program is run, these functions are called and we see the output of three functions in the order they are executed (top-down).

```
if __name__ == '__main__':
    wordBasedSentiment()
    multiWordBasedSentiment()
    markNegativity()
```

Exploring advanced sentiment analysis

We are seeing that more and more businesses are going online to increase their target customer base and the customers are given the ability to leave feedback via various channels. It's becoming more and more important for businesses to understand the emotional response of their customers with respect to the businesses they run.

In this recipe, we will write our own sentiment analysis program based on what we have learned in the previous recipe. We will also explore the built-in vader sentiment analysis algorithm, which helps evaluate in finding the sentiment of complex sentences.

Getting ready

You should have Python installed, along with the `nltk` library.

How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `AdvSentiment.py`.

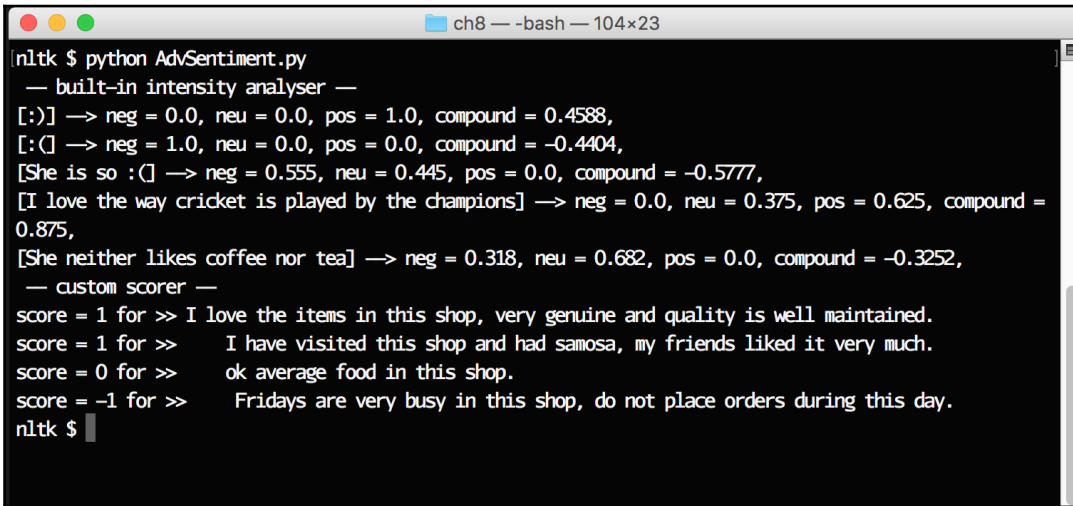
3. Type the following source code:

```

AdvSentiment.py
1 import nltk
2 import nltk.sentiment.util
3 import nltk.sentiment.sentiment_analyzer
4 from nltk.sentiment.vader import SentimentIntensityAnalyzer
5
6 def mySentimentAnalyzer():
7     def score_feedback(text):
8         positive_words = ['love', 'genuine', 'liked']
9         if '_NEG' in ' '.join(nltk.sentiment.util.mark_negation(text.split())):
10             score = -1
11         else:
12             analysis = nltk.sentiment.util.extract_unigram_feats(text.split(),
13 *             positive_words)
14             if True in analysis.values():
15                 score = 1
16             else:
17                 score = 0
18         return score
19
20     feedback = """I love the items in this shop, very genuine and quality is
21 * well maintained.
22 I have visited this shop and had samosa, my friends liked it very much.
23 ok average food in this shop.
24 Fridays are very busy in this shop, do not place orders during this day."""
25     print(' -- custom scorer --')
26     for text in feedback.split("\n"):
27         print("score = {} for >> {}".format(score_feedback(text), text))
28
29 def advancedSentimentAnalyzer():
30     sentences = [
31         ':)',
32         ':(',
33         'She is so :(',
34         'I love the way cricket is played by the champions',
35         'She neither likes coffee nor tea',
36     ]
37     senti = SentimentIntensityAnalyzer()
38     print(' -- built-in intensity analyser --')
39     for sentence in sentences:
40         print('{}{}'.format(sentence), end=' --> ')
41         kvp = senti.polarity_scores(sentence)
42         for k in kvp:
43             print('{} = {}'.format(k, kvp[k]), end='')
44         print()
45
46 if __name__ == '__main__':
47     advancedSentimentAnalyzer()
48     mySentimentAnalyzer()

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:

A terminal window titled 'ch8 — -bash — 104x23' displays the output of a Python script. The script uses NLTK's built-in intensity analyzer to score several sentences. The output shows sentiment scores for neg, neu, pos, and compound for each sentence. Below the built-in analyzer, a custom scorer is used to provide feedback scores for the same sentences.

```
nltk $ python AdvSentiment.py
— built-in intensity analyser —
[.:] → neg = 0.0, neu = 0.0, pos = 1.0, compound = 0.4588,
[:()] → neg = 1.0, neu = 0.0, pos = 0.0, compound = -0.4404,
[She is so :()] → neg = 0.555, neu = 0.445, pos = 0.0, compound = -0.5777,
[I love the way cricket is played by the champions] → neg = 0.0, neu = 0.375, pos = 0.625, compound = 0.875,
[She neither likes coffee nor tea] → neg = 0.318, neu = 0.682, pos = 0.0, compound = -0.3252,
— custom scorer —
score = 1 for >> I love the items in this shop, very genuine and quality is well maintained.
score = 1 for >> I have visited this shop and had samosa, my friends liked it very much.
score = 0 for >> ok average food in this shop.
score = -1 for >> Fridays are very busy in this shop, do not place orders during this day.
nltk $
```

How it works...

Now, let's see how our sentiment analysis program works. These four instructions import the necessary modules that we are going to use as part of this program.

```
import nltk
import nltk.sentiment.util
import nltk.sentiment.sentiment_analyzer
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

Defining a new function, `mySentimentAnalyzer()`:

```
def mySentimentAnalyzer():
```

This instruction defines a new subfunction, `score_feedback()`, which takes a sentence as input and returns the score for the sentence in terms of -1 negative, 0 neutral, and 1 positive.

```
def score_feedback(text):
```

Since we are just experimenting, we are defining the three words using which we are going to find the sentiment. In real-world use cases, we might use these from the corpus of a larger dictionary.

```
positive_words = ['love', 'genuine', 'liked']
```

This instruction breaks the input sentence into words. The list of words is fed to the `mark_negation()` function to identify the presence of any negativity in the sentence. Join the result from `mark_negation()` to the string and see if the `_NEG` suffix is present; then set the score as `-1`.

```
    if '_NEG' in '
'.join(nltk.sentiment.util.mark_negation(text.split())):
        score = -1
```

Here we are using `extract_unigram_feats()` on the input text against `positive_words` and storing the dictionary into a variable called `analysis`:

```
    else:
        analysis =
nltk.sentiment.util.extract_unigram_feats(text.split(), positive_words)
```

The value of score is decided to be 1 if there is a presence of the positive word in the input text.

```
    if True in analysis.values():
        score = 1
    else:
        score = 0
```

Finally this `score_feedback()` function returns the computed score:

```
    return score
```

These are the four reviews that we are interested in processing using our algorithm to print the score.

```
feedback = """I love the items in this shop, very genuine and quality
is well maintained.
I have visited this shop and had samosa, my friends liked it very much.
ok average food in this shop.
Fridays are very busy in this shop, do not place orders during this
day."""
```

These instructions extract the sentences from the variable `feedback` by splitting on newline (`\n`) and calling the `score_feedback()` function on this text.

```
print(' -- custom scorer --')
for text in feedback.split("\n"):
    print("score = {} for >> {}".format(score_feedback(text), text))
```

The result will be the score and sentence on the screen. This instruction defines the `advancedSentimentAnalyzer()` function, which will be used to understand the built-in features of NLTK sentiment analysis.

```
def advancedSentimentAnalyzer():
```

We are defining five sentences to analyze. you'll note that we are also using emoticons (icons) to see how the algorithm works.

```
    sentences = [
        ':)',
        ':(',
        'She is so :(',
        'I love the way cricket is played by the champions',
        'She neither likes coffee nor tea',
    ]
```

This instruction creates a new object for `SentimentIntensityAnalyzer()` and stores the object in the variable `senti`.

```
    senti = SentimentIntensityAnalyzer()

    print(' -- built-in intensity analyser --')
    for sentence in sentences:
        print('{} {}'.format(sentence), end=' --> ')
        kvp = senti.polarity_scores(sentence)
        for k in kvp:
            print('{} = {}'.format(k, kvp[k]), end='')
        print()
```

These instructions do the following things:

- Iterate over all the sentences and store the current one in the variable `sentence`
- Display the currently processed sentence on screen
- Invoke the `polarity_scores()` function on this sentence; store the result in a variable called `kvp`
- Traverse through the dictionary `kvp` and print the key (negativity, neutral, positivity, or compound types) and the score computed for these types

When the current program is invoked, call these two functions to display the results on screen.

```
if __name__ == '__main__':  
    advancedSentimentAnalyzer()  
    mySentimentAnalyzer()
```

Creating a conversational assistant or chatbot

Conversational assistants or chatbots are not very new. One of the foremost of this kind is ELIZA, which was created in the early 1960s and is worth exploring.

In order to successfully build a conversational engine, it should take care of the following things:

- Understand the target audience
- Understand the natural language in which communication happens
- Understand the intent of the user
- Come up with responses that can answer the user and give further clues

NLTK has a module, `nltk.chat`, which simplifies building these engines by providing a generic framework.

Let's see the available engines in NLTK:

Engines	Modules
Eliza	<code>nltk.chat.eliza</code> Python module
Iesha	<code>nltk.chat.iesha</code> Python module
Rude	<code>nltk.chat.rudep</code> Python module
Suntsu	<code>nltk.chat.suntsu</code> module
Zen	<code>nltk.chat.zen</code> module

In order to interact with these engines we can just load these modules in our Python program and invoke the `demo()` function.

This recipe will show us how to use built-in engines and also write our own simple conversational engine using the framework provided by the `nltk.chat` module.

Getting ready

You should have Python installed, along with the `nltk` library. Having an understanding of regular expressions also helps.

How to do it...

1. Open atom editor (or your favorite programming editor).
2. Create a new file called `Conversational.py`.

3. Type the following source code:

```

Conversational.py
1 import nltk
2
3 def builtinEngines(whichOne):
4     if whichOne == 'eliza':
5         nltk.chat.eliza.demo()
6     elif whichOne == 'iesha':
7         nltk.chat.iesha.demo()
8     elif whichOne == 'rude':
9         nltk.chat.rude.demo()
10    elif whichOne == 'suntsu':
11        nltk.chat.suntsu.demo()
12    elif whichOne == 'zen':
13        nltk.chat.zen.demo()
14    else:
15        print("unknown built-in chat engine {}".format(whichOne))
16
17 def myEngine():
18     chatpairs = (
19         (r"(.*)Stock price(.*)",
20          ("Today stock price is 100",
21           "I am unable to find out the stock price.")),
22         (r"(.*)not well(.*)",
23          ("Oh, take care. May be you should visit a doctor",
24           "Did you take some medicine ?")),
25         (r"(.*)raining(.*)",
26          ("Its monsoon season, what more do you expect ?",
27           "Yes, its good for farmers")),
28         (r"How(.*)health(.*)",
29          ("I am always healthy.",
30           "I am a program, super healthy!")),
31         (r".*",
32          ("I am good. How are you today ?",
33           "What brings you here ?"))
34     )
35
36 def chat():
37     print("!")
38     print(">> my Engine << ")
39     print("Talk to the program using normal english")
40     print("=")
41     chatbot = nltk.chat.util.Chat(chatpairs, nltk.chat.util.reflections)
42     chatbot.converse()
43
44     chat()
45
46 if __name__ == '__main__':
47     for engine in ['eliza', 'iesha', 'rude', 'suntsu', 'zen']:
48         print("=== demo of {} ===".format(engine))
49         builtinEngines(engine)
50         print()
51     myEngine()
52

```

4. Save the file.
5. Run the program using the Python interpreter.
6. You will see the following output:

```

ch8 - bash - 102x38

== demo of rude ==
Talk to the program by typing in plain English, using normal upper-
and lower-case letters and punctuation. Enter "quit" when done.

I suppose I should say hello.
>quit
I'm getting bored here. Become more interesting.

== demo of suntsu ==
Talk to the program by typing in plain English, using normal upper-
and lower-case letters and punctuation. Enter "quit" when done.

You seek enlightenment?
>quit
May victory be your future

== demo of zen ==
*****
                        Zen Chatbot!
*****
"Look beyond mere words and letters - look into your mind"
* Talk your way to truth with Zen Chatbot.
* Type 'quit' when you have had enough.
*****
Welcome, my child.
>quit
The reverse side also has a reverse side.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
>> my Engine <<
Talk to the program using normal english

Enter 'quit' when done
>stock price ?
I am unable to find out the stock price.
>quit
I am good. How are you today ?
nlTK $

```


How it works...

Let's try to understand what we are trying to achieve here. This instruction imports the `nltk` library into the current program.

```
import nltk
```

This instruction defines a new function called `builtinEngines` that takes a string parameter, `whichOne`:

```
def builtinEngines(whichOne):
```

These `if`, `elif`, `else` instructions are typical branching instructions that decide which chat engine's `demo()` function is to be invoked depending on the argument that is present in the `whichOne` variable. When the user passes an unknown engine name, it displays a message to the user that it's not aware of this engine.

```
    if whichOne == 'eliza':
        nltk.chat.eliza.demo()
    elif whichOne == 'iesha':
        nltk.chat.iesha.demo()
    elif whichOne == 'rude':
        nltk.chat.rude.demo()
    elif whichOne == 'suntsu':
        nltk.chat.suntsu.demo()
    elif whichOne == 'zen':
        nltk.chat.zen.demo()
    else:
        print("unknown built-in chat engine {}".format(whichOne))
```

It's a good practice to handle all known and unknown cases also; it makes our programs more robust in handling unknown situations.

This instruction defines a new function called `myEngine()`; this function does not take any parameters.

```
def myEngine():
```

This is a single instruction where we are defining a nested tuple data structure and assigning it to chat pairs.

```
chatpairs = (
    (r"(.*)Stock price(.*)",
     ("Today stock price is 100",
      "I am unable to find out the stock price.")),
    (r"(.*)not well(.*)",
     ("Oh, take care. May be you should visit a doctor",
      "Did you take some medicine ?")),
    (r"(.*)raining(.*)",
     ("Its monsoon season, what more do you expect ?",
      "Yes, its good for farmers")),
    (r"How(.*)health(.*)",
     ("I am always healthy.",
      "I am a program, super healthy!")),
    (r".*",
     ("I am good. How are you today ?",
      "What brings you here ?"))
)
```

Let's pay close attention to the data structure:

- We are defining a tuple of tuples
- Each subtuple consists of two elements:
 - The first member is a regular expression (this is the user's question in regex format)
 - The second member of the tuple is another set of tuples (these are the answers)

We are defining a subfunction called `chat()` inside the `myEngine()` function. This is permitted in Python. This `chat()` function displays some information to the user on the screen and calls the `nlTK` built-in `nlTK.chat.util.Chat()` class with the `chatpairs` variable. It passes `nlTK.chat.util.reflections` as the second argument. Finally we call the `chatbot.converse()` function on the object that's created using the `chat()` class.

```
def chat():
    print("!"*80)
    print(" >> my Engine << ")
    print("Talk to the program using normal english")
    print("="*80)
    print("Enter 'quit' when done")
    chatbot = nlTK.chat.util.Chat(chatpairs,
nlTK.chat.util.reflections)
    chatbot.converse()
```

This instruction calls the `chat()` function, which shows a prompt on the screen and accepts the user's requests. It shows responses according to the regular expressions that we have built before:

```
chat()
```

These instructions will be called when the program is invoked as a standalone program (not using `import`).

```
if __name__ == '__main__':
    for engine in ['eliza', 'iesha', 'rude', 'suntsu', 'zen']:
        print("=== demo of {} ===".format(engine))
        builtinEngines(engine)
        print()
    myEngine()
```

They do these two things:

- Invoke the built-in engines one after another (so that we can experience them)
- Once all the five built-in engines are excited, they call our `myEngine()`, where our customer engine comes into play

9

Applications of Deep Learning in NLP

In this chapter, we will cover the following recipes:

- Classification of emails using deep neural networks after generating TF-IDF
- IMDB sentiment classification using convolutional networks CNN 1D
- IMDB sentiment classification using bidirectional LSTM
- Visualization of high-dimensional words in 2D with neural word vector visualization

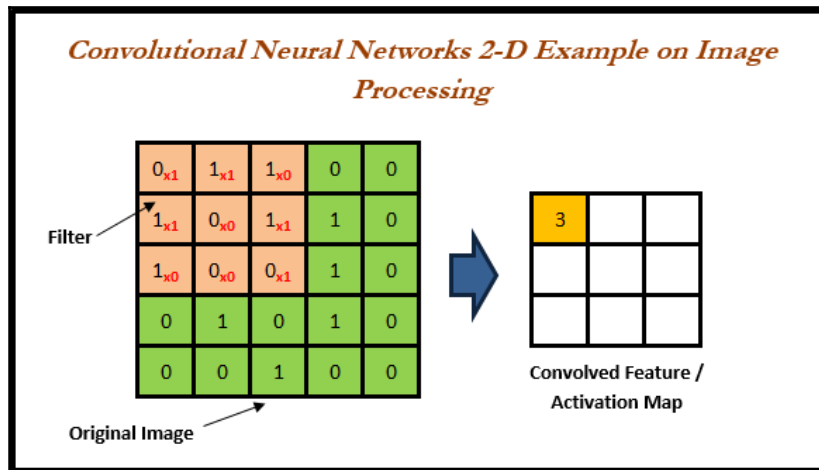
Introduction

In recent times, deep learning has become very prominent in the application of text, voice, and image data to obtain state-of-the-art results, which are primarily used in the creation of applications in the field of artificial intelligence. However, these models turn out to be producing such results in all the fields of application. In this chapter, we will be covering various applications in NLP/text processing.

Convolutional neural networks and recurrent neural networks are central themes in deep learning that you will keep meeting across the domain.

Convolutional neural networks

CNNs are primarily used in image processing to classify images into a fixed set of categories and so on. CNN's working principle has been described in the following diagram, wherein a filter of size 3×3 convolves over the original matrix of size 5×5 , which produces an output of size 3×3 . The filter can stride horizontally by a step size of 1 or any value greater than 1 also. For cell (1, 1) the value obtained is 3, which is a product of the underlying matrix value and filter values. In this way, the filter will hover across the original 5×5 matrix to create convolved features of 3×3 , also known as activation maps:



The advantages of using convolutions:

- Instead of a fixed size, fully connected layers save the number of neurons and hence the computational power requirement of the machine.
- Only a small size of filter weights is used to hover across the matrix, rather than each pixel connected to the next layers. So this is a better way of summarization of the input image into the next layers.
- During backpropagation, only the weights of the filter need to be updated based on the backpropagated errors, hence the higher efficiency.

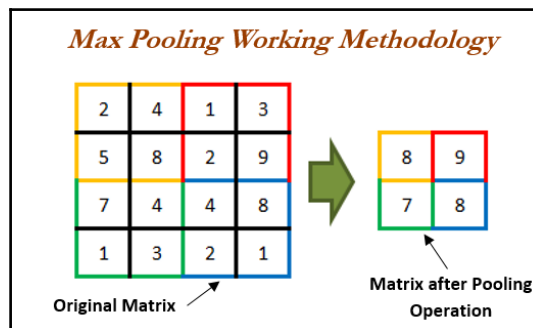
CNNs perform mappings between spatially/temporally distributed arrays in arbitrary dimensions. They appear to be suitable for application to time series, images, or videos. CNNs are characterized by:

- Translation invariance (neural weights are fixed with respect to spatial translation)

- Local connectivity (neural connections exist only between spatially local regions)
- An optional progressive decrease in spatial resolution (as the number of features is gradually increased)

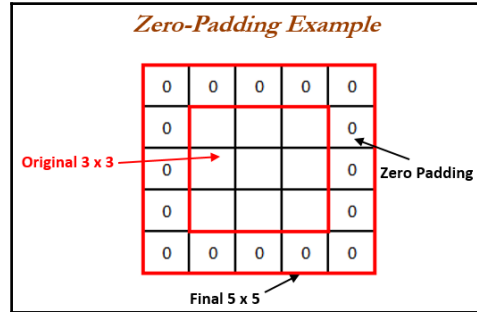
After convolution, the convolved feature/activation map needs to be reduced based on the most important features, as the same operation reduces the number of points and improves computational efficiency. Pooling is an operation typically performed to reduce unnecessary representations. Brief details about pooling operations are given as follows:

- **Pooling:** Pooling makes the activation representation (obtained from convolving the filter over the input combination of input and weight values) smaller and more manageable. It operates over each activation map independently. Pooling applies to the width and breadth of the layer, and the depth will remain the same during the pooling stage. In the following diagram, a pooling operation of 2×2 is explained. Every original 4×4 matrix has been reduced by half. In the first four cell values of 2, 4, 5, and 8, the maximum is extracted, which is 8:



Due to the operation of convolution, it is natural that the size of pixels/input data size reduces over the stages. But in some cases, we would really like to maintain the size across operations. A hacky way to achieve this is padding with zeros at the top layer accordingly.

- **Padding:** The following diagram (its width and breadth) will be shrunk consecutively; this is undesirable in deep networks, and padding keeps the size of the picture constant or controllable in size throughout the network.



A simple equation for calculating the activation map size based on given input width, filter size, padding, and stride is shown as follows. This equation gives an idea of how much computational power is needed, and so on.

- **Calculation of activation map size:** In the following formula, the size of the activation map obtained from the convolutional layer is:

$$\text{Activation Map Size} = \left(\frac{W - F + 2P}{S} \right) + 1$$

Where, W is the width of original image, F is the filter size, P is padding size (1 for a single layer of padding, 2 for a double layer of padding, and so on), S is stride length

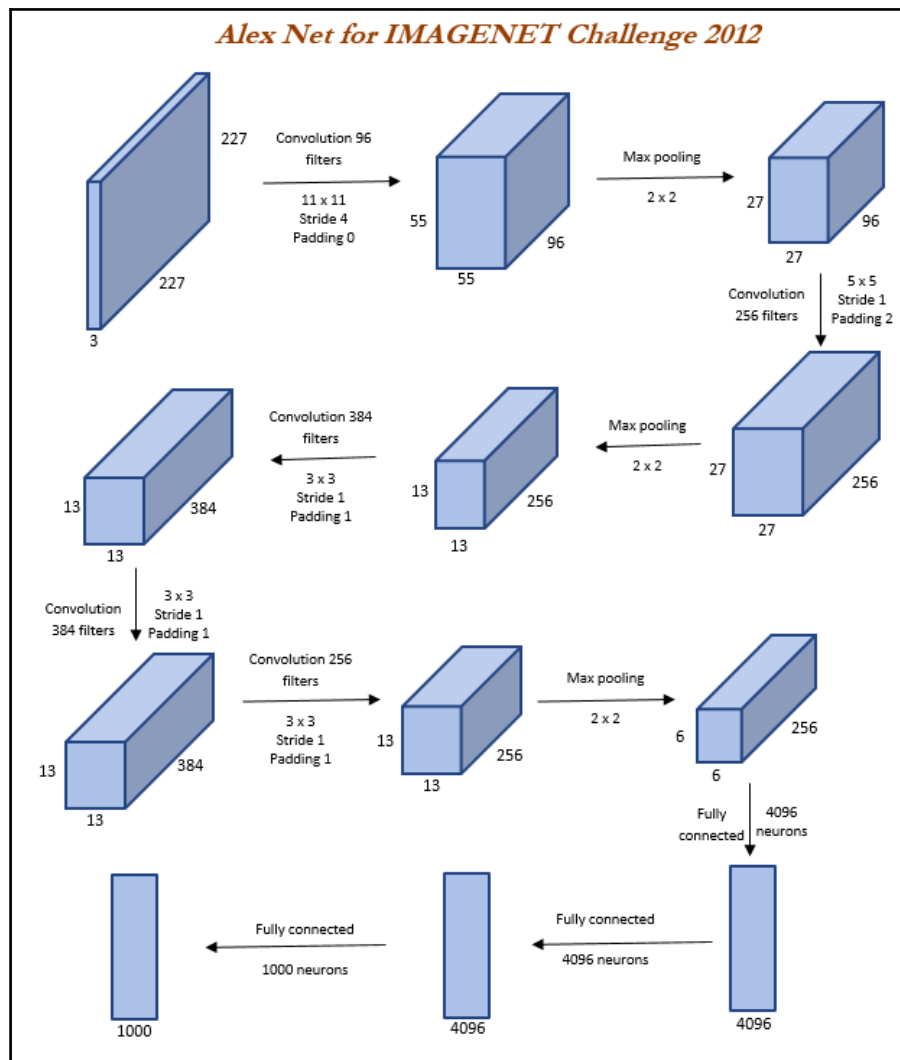
For example, consider an input image of size $224 \times 224 \times 3$ (3 indicates Red, Green, and Blue channels), with a filter size of 11×11 and number of filters as 96. The stride length is 4 and there is no padding. What is the activation map size generated out from these filters?

$$\text{Activation Map Size} = \left(\frac{W - F + 2P}{S} \right) + 1$$

$$\text{Activation Map Size} = \left(\frac{224 - 11 + 2 * 0}{4} \right) + 1 = 54.25 \sim 55$$

The activation map dimensions would be $55 \times 55 \times 96$. Using the preceding formula, only width and depth can be computed, but the depth depends on the number of filters used. In fact, this is what was obtained in step 1 after convolution stage in AlexNet, which we will describe now.

- **AlexNet used in ImageNet competition during 2012:** The following image describes AlexNet, developed to win the ImageNet competition during 2012. It produced significantly more accuracy compared with other competitors.



In AlexNet, all techniques such as convolution, pooling, and padding have been used, and finally get connected with the fully connected layer.

Applications of CNNs

CNNs are used in various applications, a few of them are as follows:

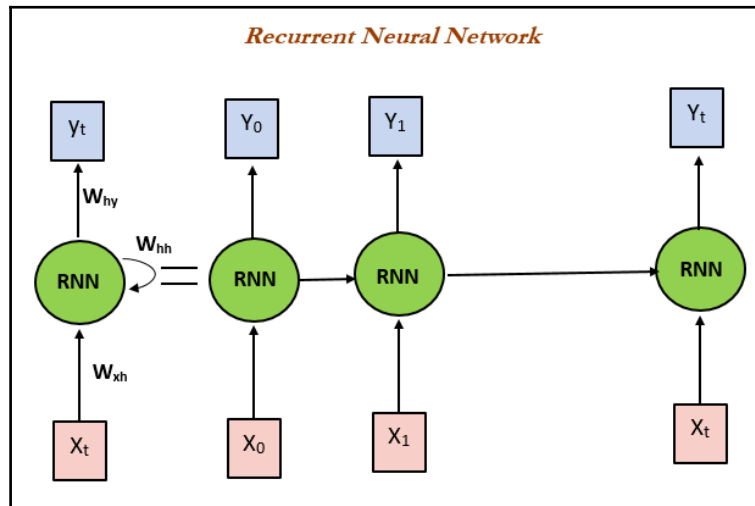
- **Image classification:** Compared with other methods, CNNs achieve higher accuracy on large-scale images of data size. In image classification, CNNs are used at the initial stage, and once enough features are extracted using pooling layers, followed by other CNNs and so on, will be finally connected with the fully connected layers to classify them into the number of given classes.
- **Face recognition:** CNNs are invariant to position, brightness, and so on, which will recognize faces from images and process them despite bad lighting, a face looking sideways, and so on.
- **Scene labeling:** Each pixel is labeled with the category of the object it belongs to in scene labeling. CNNs are utilized here to combine pixels in a hierarchical manner.
- **NLP:** In NLP, CNNs are used similarly with bag-of-words, in which the sequence of words does not play a critical role in identifying the final class of email/text and so on. CNNs are used on matrices, which are represented by sentences in vector format. Subsequently, filters are applied but CNNs are one-dimensional, in which width is constant, and filters traverse only across height (the height is 2 for bi-grams, 3 for tri-grams, and so on).

Recurrent neural networks

A recurrent neural network is used to process a sequence of vectors X by applying a recurrence formula at every time step. In convolutional neural networks, we assume all inputs are independent of each other. But in some tasks, inputs are dependent on each other, for example, time series forecasting data, or predicting the next word in a sentence depending on past words, and so on, which needs to be modeled by considering dependency of past sequences. These types of problems are modeled with RNNs as they provide better accuracy. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only for a few steps. The next formula explains the RNN functionality:

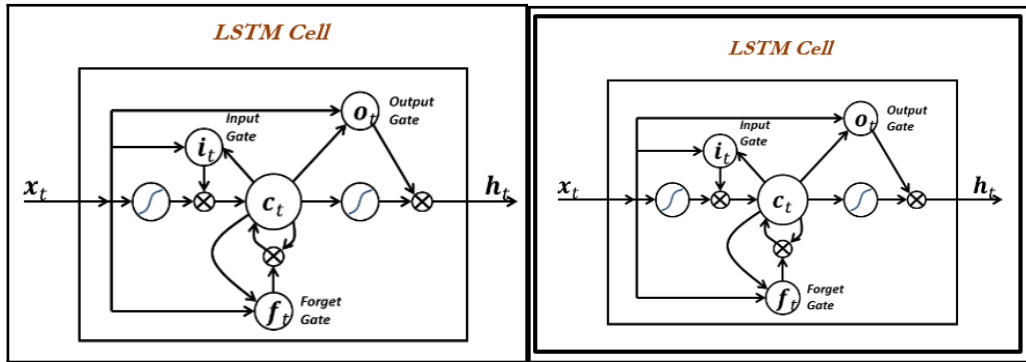
$$h_t = f_W(h_{t-1}, x_t)$$

$$h_{t-1} = \text{old state}; x_t = \text{input vector at some time step}$$



- Vanishing or exploding the gradient problem in RNNs:** Gradients does vanish quickly with the more number of layers and this issue is severe with RNNs as at each layer there are many time steps which also do occur and recurrent weights are multiplicative in nature, hence gradients either explode or vanish quickly, which makes neural networks untrainable. Exploding gradients can be limited by using a gradient clipping technique, in which an upper limit will be set to explode the gradients, but however vanishing gradient problem still does exists. This issue can be overcome by using **long short-term memory (LSTM)** networks.

- **LSTM:** LSTM is an artificial neural network contains LSTM blocks in addition to regular network units. LSTM blocks contain gates that determine when the input is significant enough to remember, when it should continue to remember or when it should forget the value, and when it should output the value.



Vanishing and exploding gradient problems do not occur in LSTM as the same is an additive model rather than multiplicative model which is the case with RNN.

Application of RNNs in NLP

RNNs have shown great success in many NLP tasks. The most commonly used variant of RNN is LSTM due to overcoming the issue of vanishing/exploding gradients.

- **Language modeling:** Given a sequence of words, the task is to predict the next probable word
- **Text generation:** To generate text from the writings of some authors
- **Machine translation:** To convert one language into other language (English to Chinese and so on.)
- **Chat bot:** This application is very much like machine translation; however question and answer pairs are used to train the model
- **Generating an image description:** By training together with CNNs, RNNs can be used to generate a caption/description of the image

Classification of emails using deep neural networks after generating TF-IDF

In this recipe, we will use deep neural networks to classify emails into one of the 20 pre-trained categories based on the words present in each email. This is the simple model to start with to understand the subject of deep learning and its applications on NLP.

Getting ready

The 20 newsgroups dataset from scikit-learn have been utilized to illustrate the concept. Number of observations/emails considered for analysis are 18,846 (train observations - 11,314 and test observations - 7,532) and its corresponding classes/categories are 20, which are shown in the following:

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> newsgroups_train = fetch_20newsgroups(subset='train')
>>> newsgroups_test = fetch_20newsgroups(subset='test')
>>> x_train = newsgroups_train.data
>>> x_test = newsgroups_test.data
>>> y_train = newsgroups_train.target
>>> y_test = newsgroups_test.target
>>> print ("List of all 20 categories:")
>>> print (newsgroups_train.target_names)
>>> print ("\n")
>>> print ("Sample Email:")
>>> print (x_train[0])
>>> print ("Sample Target Category:")
>>> print (y_train[0])
>>> print (newsgroups_train.target_names[y_train[0]])
```

In the following screenshot, a sample first data observation and target class category has been shown. From the first observation or email we can infer that the email is talking about a two-door sports car, which we can classify manually into autos category which is 8.



Target value is 7 due to the indexing starts from 0), which is validating our understanding with actual target class 7

```
List of all 20 categories:
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc', 'talk.religion.misc']
```

```
Sample Email:
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15
```

```
I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.
```

```
Thanks,
- IL
    ---- brought to you by your neighborhood Lerxst ----
```

```
Sample Target Category:
7
rec.autos
```

How to do it...

Using NLP techniques, we have pre-processed the data for obtaining finalized word vectors to map with final outcomes spam or ham. Major steps involved are:

1. Pre-processing.
2. Removal of punctuations.
3. Word tokenization.
4. Converting words into lowercase.
5. Stop word removal.

6. Keeping words of length of at least 3.
7. Stemming words.
8. POS tagging.
9. Lemmatization of words:
 1. TF-IDF vector conversion.
 2. Deep learning model training and testing.
 3. Model evaluation and results discussion.

How it works...

The NLTK package has been utilized for all the pre-processing steps, as it consists of all the necessary NLP functionality under one single roof:

```
# Used for pre-processing data
>>> import nltk
>>> from nltk.corpus import stopwords
>>> from nltk.stem import WordNetLemmatizer
>>> import string
>>> import pandas as pd
>>> from nltk import pos_tag
>>> from nltk.stem import PorterStemmer
```

The function written (pre-processing) consists of all the steps for convenience. However, we will be explaining all the steps in each section:

```
>>> def preprocessing(text):
```

The following line of the code splits the word and checks each character to see if it contains any standard punctuations, if so it will be replaced with a blank or else it just don't replace with blank:

```
...     text2 = " ".join("".join([" " if ch in string.punctuation else ch
for ch in text]).split())
```

The following code tokenizes the sentences into words based on whitespaces and puts them together as a list for applying further steps:

```
...     tokens = [word for sent in nltk.sent_tokenize(text2) for word in
nltk.word_tokenize(sent)]
```

Converting all the cases (upper, lower and proper) into lower case reduces duplicates in corpus:

```
...     tokens = [word.lower() for word in tokens]
```

As mentioned earlier, Stop words are the words that do not carry much of weight in understanding the sentence; they are used for connecting words and so on. We have removed them with the following line of code:

```
... stopwds = stopwords.words('english')
... tokens = [token for token in tokens if token not in stopwds]
```

Keeping only the words with length greater than 3 in the following code for removing small words which hardly consists of much of a meaning to carry;

```
... tokens = [word for word in tokens if len(word)>=3]
```

Stemming applied on the words using Porter stemmer which stems the extra suffixes from the words:

```
... stemmer = PorterStemmer()
... tokens = [stemmer.stem(word) for word in tokens]
```

POS tagging is a prerequisite for lemmatization, based on whether word is noun or verb or and so on. it will reduce it to the root word

```
... tagged_corpus = pos_tag(tokens)
```

pos_tag function returns the part of speech in four formats for Noun and six formats for verb. NN - (noun, common, singular), NNP - (noun, proper, singular), NNPS - (noun, proper, plural), NNS - (noun, common, plural), VB - (verb, base form), VBD - (verb, past tense), VBG - (verb, present participle), VBN - (verb, past participle), VBP - (verb, present tense, not 3rd person singular), VBZ - (verb, present tense, third person singular)

```
... Noun_tags = ['NN', 'NNP', 'NNPS', 'NNS']
... Verb_tags = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
... lemmatizer = WordNetLemmatizer()
```

The following function, prat_lemmatize, has been created only for the reasons of mismatch between the pos_tag function and intake values of lemmatize function. If the tag for any word falls under the respective noun or verb tags category, n or v will be applied accordingly in lemmatize function:

```
... def prat_lemmatize(token,tag):
...     if tag in Noun_tags:
...         return lemmatizer.lemmatize(token,'n')
...     elif tag in Verb_tags:
...         return lemmatizer.lemmatize(token,'v')
...     else:
...         return lemmatizer.lemmatize(token,'n')
```

After performing tokenization and applied all the various operations, we need to join it back to form strings and the following function performs the same:

```
...     pre_proc_text = " ".join([prat_lemmatize(token,tag) for token,tag
in tagged_corpus])
...     return pre_proc_text
```

Applying pre-processing on train and test data:

```
>>> x_train_preprocessed = []
>>> for i in x_train:
...     x_train_preprocessed.append(preprocessing(i))
>>> x_test_preprocessed = []
>>> for i in x_test:
...     x_test_preprocessed.append(preprocessing(i))
# building TFIDF vectorizer
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer(min_df=2, ngram_range=(1, 2),
stop_words='english', max_features= 10000,strip_accents='unicode',
norm='l2')
>>> x_train_2 = vectorizer.fit_transform(x_train_preprocessed).todense()
>>> x_test_2 = vectorizer.transform(x_test_preprocessed).todense()
```

After the pre-processing step has been completed, processed TF-IDF vectors have to be sent to the following deep learning code:

```
# Deep Learning modules
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers.core import Dense, Dropout, Activation
>>> from keras.optimizers import Adadelta,Adam,RMSprop
>>> from keras.utils import np_utils
```

The following image produces the output after firing up the preceding Keras code. Keras has been installed on Theano, which eventually works on Python. A GPU with 6 GB memory has been installed with additional libraries (CuDNN and CNMeM) for four to five times faster execution, with a choking of around 20% memory; hence only 80% memory out of 6 GB is available;

```
Using Theano backend.
WARNING (theano.sandbox.cuda): The cuda backend is deprecated and will be removed in the next release (v0.10). Please s
witch to the gpuarray backend. You can get more information about how to switch at this URL:
https://github.com/Theano/Theano/wiki/Converting-to-the-new-gpu-back-end%28gpuarray%29
Using gpu device 0: GeForce GTX 1060 6GB (CNMeM is enabled with initial size: 80.0% of memory, cuDNN 5105)
```


The following code explains the central part of the deep learning model. The code is self-explanatory, with the number of classes considered 20, batch size 64, and number of epochs to train, 20:

```
# Definition hyper parameters
>>> np.random.seed(1337)
>>> nb_classes = 20
>>> batch_size = 64
>>> nb_epochs = 20
```

The following code converts the 20 categories into one-hot encoding vectors in which 20 columns are created and the values against the respective classes are given as 1. All other classes are given as 0:

```
>>> Y_train = np_utils.to_categorical(y_train, nb_classes)
```

In the following building blocks of Keras code, three hidden layers (1000, 500, and 50 neurons in each layer respectively) are used, with dropout as 50% for each layer with Adam as an optimizer:

```
#Deep Layer Model building in Keras
#del model
>>> model = Sequential()
>>> model.add(Dense(1000,input_shape= (10000,)))
>>> model.add(Activation('relu'))
>>> model.add(Dropout(0.5))
>>> model.add(Dense(500))
>>> model.add(Activation('relu'))
>>> model.add(Dropout(0.5))
>>> model.add(Dense(50))
>>> model.add(Activation('relu'))
>>> model.add(Dropout(0.5))
>>> model.add(Dense(nb_classes))
>>> model.add(Activation('softmax'))
>>> model.compile(loss='categorical_crossentropy', optimizer='adam')
>>> print (model.summary())
```

The architecture is shown as follows and describes the flow of the data from a start of 10,000 as input. Then there are 1000, 500, 50, and 20 neurons to classify the given email into one of the 20 categories:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 1000)	10001000
activation_1 (Activation)	(None, 1000)	0
dropout_1 (Dropout)	(None, 1000)	0
dense_2 (Dense)	(None, 500)	500500
activation_2 (Activation)	(None, 500)	0
dropout_2 (Dropout)	(None, 500)	0
dense_3 (Dense)	(None, 50)	25050
activation_3 (Activation)	(None, 50)	0
dropout_3 (Dropout)	(None, 50)	0
dense_4 (Dense)	(None, 20)	1020
activation_4 (Activation)	(None, 20)	0
=====		
Total params: 10,527,570.0		
Trainable params: 10,527,570.0		
Non-trainable params: 0.0		

The model is trained as per the given metrics:

```
# Model Training
>>> model.fit(x_train_2, Y_train, batch_size=batch_size,
epochs=nb_epochs, verbose=1)
```

The model has been fitted with 20 epochs, in which each epoch took about 2 seconds. The loss has been minimized from 1.9281 to 0.0241. By using CPU hardware, the time required for training each epoch may increase as a GPU massively parallelizes the computation with thousands of threads/cores:

```
Epoch 1/20
11314/11314 [=====] - 2s - loss: 1.9281
Epoch 2/20
11314/11314 [=====] - 2s - loss: 0.5844
Epoch 3/20
11314/11314 [=====] - 2s - loss: 0.2854
Epoch 4/20
11314/11314 [=====] - 2s - loss: 0.1709
Epoch 17/20
11314/11314 [=====] - 2s - loss: 0.0218
Epoch 18/20
11314/11314 [=====] - 2s - loss: 0.0217
Epoch 19/20
11314/11314 [=====] - 2s - loss: 0.0229
Epoch 20/20
11314/11314 [=====] - 2s - loss: 0.0241
Out[13]: <keras.callbacks.History at 0x1701c0f60>
```

Finally, predictions are made on the train and test datasets to determine the accuracy, precision, and recall values:

```
#Model Prediction
>>> y_train_predclass =
model.predict_classes(x_train_2,batch_size=batch_size)
>>> y_test_predclass =
model.predict_classes(x_test_2,batch_size=batch_size)
>>> from sklearn.metrics import accuracy_score,classification_report
>>> print ("\n\nDeep Neural Network - Train
accuracy:"),(round(accuracy_score( y_train, y_train_predclass),3))
>>> print ("\nDeep Neural Network - Test accuracy:"),(round(accuracy_score(
y_test,y_test_predclass),3))
>>> print ("\nDeep Neural Network - Train Classification Report")
>>> print (classification_report(y_train,y_train_predclass))
>>> print ("\nDeep Neural Network - Test Classification Report")
>>> print (classification_report(y_test,y_test_predclass))
```

Deep Neural Network - Train accuracy: 0.999				
Deep Neural Network - Test accuracy: 0.807				
Deep Neural Network - Train Classification	Report			
precision	recall	f1-score	support	
0	1.00	1.00	1.00	480
1	0.99	1.00	1.00	584
2	1.00	1.00	1.00	591
3	1.00	1.00	1.00	590
4	1.00	1.00	1.00	578
5	1.00	1.00	1.00	593
6	1.00	1.00	1.00	585
7	1.00	1.00	1.00	594
8	1.00	1.00	1.00	598
9	1.00	1.00	1.00	597
10	1.00	1.00	1.00	600
11	1.00	1.00	1.00	595
12	1.00	1.00	1.00	591
13	1.00	1.00	1.00	594
14	1.00	1.00	1.00	593
15	1.00	1.00	1.00	599
16	1.00	1.00	1.00	546
17	1.00	1.00	1.00	564
18	1.00	1.00	1.00	465
19	1.00	1.00	1.00	377
avg / total	1.00	1.00	1.00	11314
Deep Neural Network - Test Classification				
precision	recall	f1-score	Report	support
0	0.74	0.73	0.74	319
1	0.61	0.75	0.67	389
2	0.74	0.69	0.71	394
3	0.71	0.67	0.69	392
4	0.76	0.78	0.77	385
5	0.86	0.76	0.81	395
6	0.85	0.80	0.82	390
7	0.89	0.84	0.86	396
8	0.94	0.91	0.92	398
9	0.91	0.89	0.90	397
10	0.94	0.97	0.96	399
11	0.92	0.91	0.91	396
12	0.64	0.75	0.69	393
13	0.92	0.80	0.86	396
14	0.92	0.89	0.91	394
15	0.81	0.89	0.85	398
16	0.75	0.89	0.81	364
17	0.94	0.82	0.88	376
18	0.77	0.64	0.70	310
19	0.55	0.65	0.60	251
avg / total	0.81	0.81	0.81	7532

It appears that the classifier is giving a good 99.9% accuracy on the train dataset and 80.7% on the test dataset.

IMDB sentiment classification using convolutional networks CNN 1D

In this recipe, we will use the Keras IMDB movie review sentiment data, which has labeled its sentiment (positive/negative). Reviews are pre-processed, and each review is already encoded as a sequence of word indexes (integers). However, we have decoded it to show a you sample in the following code.

Getting ready

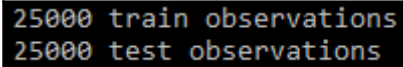
The IMDB dataset from Keras has a set of words and its respective sentiment. The following is the pre-processing of the data:

```
>>> import pandas as pd
>>> from keras.preprocessing import sequence
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Activation
>>> from keras.layers import Embedding
>>> from keras.layers import Conv1D, GlobalMaxPooling1D
>>> from keras.datasets import imdb
>>> from sklearn.metrics import accuracy_score, classification_report
```

In this set of parameters, we did put maximum features or number of words to be extracted are 6,000 with maximum length of an individual sentence as 400 words:

```
# set parameters:
>>> max_features = 6000
>>> max_length = 400
>>> (x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=max_features)
>>> print(len(x_train), 'train observations')
>>> print(len(x_test), 'test observations')
```

The dataset has an equal number of train and test observations, in which we will build a model on 25,000 observations and test the trained model on the test data with 25,000 data observations. A sample of data can be seen in this screenshot:

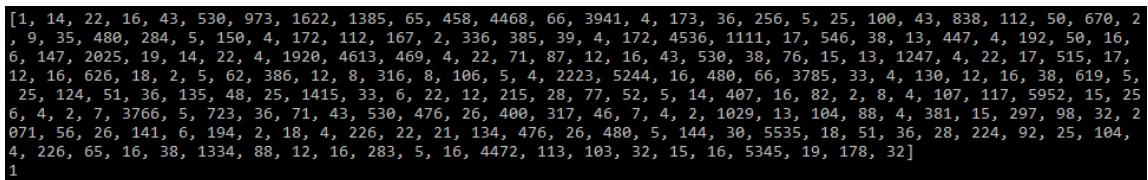


```
25000 train observations
25000 test observations
```

The following code is used to create the dictionary mapping of a word and its respective integer index value:

```
# Creating numbers to word mapping
>>> wind = imdb.get_word_index()
>>> revind = dict((v,k) for k,v in wind.iteritems())
>>> print (x_train[0])
>>> print (y_train[0])
```

We see the first observation as a set of numbers rather than any English word, because the computer can only understand and work with numbers rather than characters, words, and so on:



```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 25, 6, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2, 071, 56, 26, 141, 6, 194, 2, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345, 19, 178, 32]
1
```

Decoding using a created dictionary of inverse mapping is shown here:

```
>>> def decode(sent_list):
...     new_words = []
...     for i in sent_list:
...         new_words.append(revind[i])
...     comb_words = " ".join(new_words)
...     return comb_words
>>> print (decode(x_train[0]))
```

The following screenshot describes the stage after converting a number mapping into textual format. Here, dictionaries are utilized to reverse a map from integer format to text format:

```
the as you with out themselves powerful lets loves their becomes reaching had journalist of lot from anyone to have after
out atmosphere never more room and it so heart shows to years of every never going and help moments or of every chest vi
sual movie except her was several of enough more with is now current film as you of mine potentially unfortunately of you
than him that with out themselves her get for was camp of you movie sometimes movie that with scary but and to story won
derful that in seeing in character to of 70s musicians with heart had shadows they of here that with her serious to have
does when from why what have critics they is you that isn't one will very to as itself with other and in of seen over lan
ded for anyone of and br show's to whether from than out themselves history he name half some br of and odd was two most
of mean for 1 any an boat she he should is thought and but of script you not while history he heart to real at barrel but
when from one bit then have two of script their with her nobody most that with wasn't to with armed acting watch an for
with heartfelt film want an
```

How to do it...

The major steps involved are described as follows:

1. Pre-processing, during this stage, we do pad sequences to bring all observations into one fixed dimension, which enhances speed and enables computation.
2. CNN 1D model development and validation.
3. Model evaluation.

How it works...

The following code does perform padding operation for adding extra sentences which can make up to maximum length of 400 words. By doing this, data will become even and easier to perform computation using neural networks:

```
#Pad sequences for computational efficiency
>>> x_train = sequence.pad_sequences(x_train, maxlen=max_length)
>>> x_test = sequence.pad_sequences(x_test, maxlen=max_length)
>>> print('x_train shape:', x_train.shape)
>>> print('x_test shape:', x_test.shape)
```

```
x_train shape: (25000L, 400L)
x_test shape: (25000L, 400L)
```

The following deep learning code describes the application of Keras code to create a CNN 1D model:

```
# Deep Learning architecture parameters
>>> batch_size = 32
>>> embedding_dims = 60
```

```

>>> num_kernels = 260
>>> kernel_size = 3
>>> hidden_dims = 300
>>> epochs = 3
# Building the model
>>> model = Sequential()
>>> model.add(Embedding(max_features, embedding_dims, input_length=
max_length))
>>> model.add(Dropout(0.2))
>>> model.add(Conv1D(num_kernels, kernel_size, padding='valid',
activation='relu', strides=1))
>>> model.add(GlobalMaxPooling1D())
>>> model.add(Dense(hidden_dims))
>>> model.add(Dropout(0.5))
>>> model.add(Activation('relu'))
>>> model.add(Dense(1))
>>> model.add(Activation('sigmoid'))
>>> model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
>>> print (model.summary())

```

In the following screenshot, the entire model summary has been displayed, indicating the number of dimensions and its respective number of neurons utilized. These directly impact the number of parameters that will be utilized in computation from input data into the final target variable, whether it is 0 or 1. Hence a dense layer has been utilized at the last layer of the network:

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 400, 60)	360000
dropout_1 (Dropout)	(None, 400, 60)	0
conv1d_1 (Conv1D)	(None, 398, 260)	47060
global_max_pooling1d_1 (Glob	(None, 260)	0
dense_1 (Dense)	(None, 300)	78300
dropout_2 (Dropout)	(None, 300)	0
activation_1 (Activation)	(None, 300)	0
dense_2 (Dense)	(None, 1)	301
activation_2 (Activation)	(None, 1)	0
Total params: 485,661.0		
Trainable params: 485,661.0		
Non-trainable params: 0.0		
None		

The following code performs model fitting operation on training data in which both x and y variables are used to train data by batch wise:

```
>>> model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
              validation_split=0.2)
```

The model has been trained for three epochs, in which each epoch consumes 5 seconds on the GPU. But if we observe the following iterations, even though the train accuracy is moving up, validation accuracy is decreasing. This phenomenon can be identified as model overfitting. This indicates that we need to try some other ways to improve the model accuracy rather than just increase the number of epochs. Other ways we probably should look at are increasing the architecture size and so on. Readers are encouraged to experiment with various combinations.

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/3
20000/20000 [=====] - 5s - loss: 0.4321 - acc: 0.7872 - val_loss: 0.2896 - val_acc: 0.8750
Epoch 2/3
20000/20000 [=====] - 5s - loss: 0.2498 - acc: 0.9001 - val_loss: 0.2890 - val_acc: 0.8802
Epoch 3/3
20000/20000 [=====] - 5s - loss: 0.1635 - acc: 0.9397 - val_loss: 0.2875 - val_acc: 0.8836
Out[3]: <keras.callbacks.History at 0x1668d4f28>
```

The following code is used for prediction of classes for both train and test data:

```
#Model Prediction
>>> y_train_predclass =
model.predict_classes(x_train, batch_size=batch_size)
>>> y_test_predclass = model.predict_classes(x_test, batch_size=batch_size)
>>> y_train_predclass.shape = y_train.shape
>>> y_test_predclass.shape = y_test.shape

# Model accuracies and metrics calculation
>>> print ((" \n \n CNN 1D - Train accuracy: "), (round(accuracy_score(y_train,
y_train_predclass), 3)))
>>> print (" \n CNN 1D of Training data \n ", classification_report(y_train,
y_train_predclass))
>>> print (" \n CNN 1D - Train Confusion Matrix \n \n ", pd.crosstab(y_train,
y_train_predclass, rownames = ["Actual1"], colnames = ["Predicted"]))
>>> print ((" \n CNN 1D - Test accuracy: "), (round(accuracy_score(y_test,
y_test_predclass), 3)))
>>> print (" \n CNN 1D of Test data \n ", classification_report(y_test,
y_test_predclass))
>>> print (" \n CNN 1D - Test Confusion Matrix \n \n ", pd.crosstab(y_test,
y_test_predclass, rownames = ["Actual1"], colnames = ["Predicted"])))
```

The following screenshot describes various measurable metrics to judge the model performance. From the result, the train accuracy seems significantly high at 96%; however, the test accuracy is at a somewhat lower value of 88.2 %. This could be due to model overfitting:

```

CNN 1D - Train accuracy: 0.96

CNN 1D of Training data
precision    recall  f1-score   support

      0      0.97      0.95      0.96     12500
      1      0.95      0.97      0.96     12500

avg / total      0.96      0.96      0.96     25000

CNN 1D - Train Confusion Matrix

Predicted      0      1
Actual
0      11825      675
1       319     12181

CNN 1D - Test accuracy: 0.882

CNN 1D of Test data
precision    recall  f1-score   support

      0      0.90      0.86      0.88     12500
      1      0.86      0.91      0.89     12500

avg / total      0.88      0.88      0.88     25000

CNN 1D - Test Confusion Matrix

Predicted      0      1
Actual
0      10689     1811
1       1139     11361

```

IMDB sentiment classification using bidirectional LSTM

In this recipe, we are using same IMDB sentiment data to show the difference between CNN and RNN methodology in terms of accuracies and so on. Data pre-processing steps remain the same; only the architecture of the model varies.

Getting ready

The IMDB dataset from Keras has set of words and its respective sentiment. Here is the pre-processing of the data:

```
>>> from __future__ import print_function
>>> import numpy as np
>>> import pandas as pd
>>> from keras.preprocessing import sequence
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Embedding, LSTM, Bidirectional
>>> from keras.datasets import imdb
>>> from sklearn.metrics import accuracy_score, classification_report

# Max features are limited
>>> max_features = 15000
>>> max_len = 300
>>> batch_size = 64

# Loading data
>>> (x_train, y_train), (x_test, y_test) =
imdb.load_data(num_words=max_features)
>>> print(len(x_train), 'train observations')
>>> print(len(x_test), 'test observations')
```

How to do it...

The major steps involved are described as follows:

1. Pre-processing, during this stage, we do pad sequences to bring all the observations into one fixed dimension, which enhances speed and enables computation.
2. LSTM model development and validation.
3. Model evaluation.

How it works...

```
# Pad sequences for computational efficiency
>>> x_train_2 = sequence.pad_sequences(x_train, maxlen=max_len)
>>> x_test_2 = sequence.pad_sequences(x_test, maxlen=max_len)
>>> print('x_train shape:', x_train_2.shape)
>>> print('x_test shape:', x_test_2.shape)
>>> y_train = np.array(y_train)
>>> y_test = np.array(y_test)
```

The following deep learning code describes the application of Keras code to create a bidirectional LSTM model:

Bidirectional LSTMs have a connection from both forward and backward, which enables them to fill in the middle words to get connected well with left and right words:

```
# Model Building
>>> model = Sequential()
>>> model.add(Embedding(max_features, 128, input_length=max_len))
>>> model.add(Bidirectional(LSTM(64)))
>>> model.add(Dropout(0.5))
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])
# Print model architecture
>>> print(model.summary())
```

Here is the architecture of the model. The embedding layer has been used to reduce the dimensions to 128, followed by bidirectional LSTM, ending up with a dense layer for modeling sentiment either zero or one:

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 300, 128)	1920000
bidirectional_1 (Bidirection	(None, 128)	98816
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129
Total params: 2,018,945.0		
Trainable params: 2,018,945.0		
Non-trainable params: 0.0		
None		

The following code is used for training the data:

```
#Train the model
>>> model.fit(x_train_2, y_train, batch_size=batch_size, epochs=4,
              validation_split=0.2)
```

LSTM models take longer than CNNs because LSTMs are not easily parallelizable with GPU (4x to 5x), whereas CNNs (100x) are massively parallelizable. One important observation: even after an increase in the training accuracy, the validation accuracy was decreasing. This situation indicates overfitting.

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/4
20000/20000 [=====] - 205s - loss: 0.4366 - acc: 0.7936 - val_loss: 0.3239 - val_acc: 0.8656
Epoch 2/4
20000/20000 [=====] - 205s - loss: 0.2352 - acc: 0.9128 - val_loss: 0.3779 - val_acc: 0.8676
Epoch 3/4
20000/20000 [=====] - 205s - loss: 0.1661 - acc: 0.9426 - val_loss: 0.3661 - val_acc: 0.8664
Epoch 4/4
20000/20000 [=====] - 203s - loss: 0.1102 - acc: 0.9626 - val_loss: 0.3887 - val_acc: 0.8630
Out[5]: <keras.callbacks.History at 0x167954d30>
```

The following code has been used for predicting the class for both train and test data:

```
#Model Prediction
>>> y_train_predclass = model.predict_classes(x_train_2, batch_size=1000)
>>> y_test_predclass = model.predict_classes(x_test_2, batch_size=1000)
>>> y_train_predclass.shape = y_train.shape
>>> y_test_predclass.shape = y_test.shape

# Model accuracies and metrics calculation
>>> print ("\n\nLSTM Bidirectional Sentiment Classification - Train
accuracy: ", (round(accuracy_score(y_train, y_train_predclass), 3)))
>>> print ("\n\nLSTM Bidirectional Sentiment Classification of Training
data\n", classification_report(y_train, y_train_predclass))
>>> print ("\n\nLSTM Bidirectional Sentiment Classification - Train Confusion
Matrix\n\n", pd.crosstab(y_train, y_train_predclass, rownames =
["Actual1"], colnames = ["Predicted"]))
>>> print ("\n\nLSTM Bidirectional Sentiment Classification - Test
accuracy: ", (round(accuracy_score(y_test, y_test_predclass), 3)))
>>> print ("\n\nLSTM Bidirectional Sentiment Classification of Test
data\n", classification_report(y_test, y_test_predclass))
>>> print ("\n\nLSTM Bidirectional Sentiment Classification - Test Confusion
Matrix\n\n", pd.crosstab(y_test, y_test_predclass, rownames =
["Actual1"], colnames = ["Predicted"]))
```

```

LSTM Bidirectional Sentiment Classification - Train accuracy: 0.957

LSTM Bidirectional Sentiment Classification of Training data
      precision    recall  f1-score   support

      0         0.95      0.97      0.96      12500
      1         0.97      0.94      0.96      12500

 avg / total         0.96      0.96      0.96      25000


LSTM Bidirectional Sentiment Classification - Train Confusion Matrix

Predicted      0      1
Actual\
0      12124      376
1       700    11800


LSTM Bidirectional Sentiment Classification - Test accuracy: 0.856

LSTM Bidirectional Sentiment Classification of Test data
      precision    recall  f1-score   support

      0         0.83      0.89      0.86      12500
      1         0.88      0.82      0.85      12500

 avg / total         0.86      0.86      0.86      25000


LSTM Bidirectional Sentiment Classification - Test Confusion Matrix

Predicted      0      1
Actual\
0      11140      1360
1       2242     10258

```

It appears that LSTM did provide slightly less test accuracy compared with CNN; however, with careful tuning of the model parameters, we can obtain better accuracies in RNNs compared with CNNs.

Visualization of high-dimensional words in 2D with neural word vector visualization

In this recipe, we will use deep neural networks to visualize words from a high-dimensional space in a two-dimensional space.

Getting ready

The *Alice in Wonderland* dataset has been used to extract words and create a visualization using the dense network made to be like the encoder-decoder architecture:

```
>>> from __future__ import print_function
>>> import os
""" First change the following directory link to where all input files do
exist """
>>> os.chdir("C:\\Users\\prata\\Documents\\book_codes\\NLP_DL")
>>> import nltk
>>> from nltk.corpus import stopwords
>>> from nltk.stem import WordNetLemmatizer
>>> from nltk import pos_tag
>>> from nltk.stem import PorterStemmer
>>> import string
>>> import numpy as np
>>> import pandas as pd
>>> import random
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import OneHotEncoder
>>> import matplotlib.pyplot as plt
>>> def preprocessing(text):
... text2 = " ".join(" ".join([" " if ch in string.punctuation else ch for
ch in text]).split())
... tokens = [word for sent in nltk.sent_tokenize(text2) for word in
nltk.word_tokenize(sent)]
... tokens = [word.lower() for word in tokens]
... stopwds = stopwords.words('english')
... tokens = [token for token in tokens if token not in stopwds]
... tokens = [word for word in tokens if len(word)>=3]
... stemmer = PorterStemmer()
... tokens = [stemmer.stem(word) for word in tokens]
... tagged_corpus = pos_tag(tokens)
... Noun_tags = ['NN', 'NNP', 'NNPS', 'NNS']
... Verb_tags = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
... lemmatizer = WordNetLemmatizer()
... def prat_lemmatize(token,tag):
... if tag in Noun_tags:
... return lemmatizer.lemmatize(token, 'n')
... elif tag in Verb_tags:
... return lemmatizer.lemmatize(token, 'v')
... else:
... return lemmatizer.lemmatize(token, 'n')
... pre_proc_text = " ".join([prat_lemmatize(token,tag) for token,tag in
tagged_corpus])
... return pre_proc_text
>>> lines = []
```

```
>>> fin = open("alice_in_wonderland.txt", "rb")
>>> for line in fin:
...     line = line.strip().decode("ascii", "ignore").encode("utf-8")
...     if len(line) == 0:
...         continue
...     lines.append(preprocessing(line))
>>> fin.close()
```

How to do it...

The major steps involved are described here:

- Pre-processing, creation of skip-grams and using the middle word to predict either the left or the right word.
- Application of one-hot encoding for feature engineering.
- Model building using encoder-decoder architecture.
- Extraction of the encoder architecture to create two-dimensional features for visualization from test data.

How it works...

The following code creates dictionary, which is a mapping of word to index and index to word (vice versa). As we knew, models simply do not work on character/word input. Hence, we will be converting words into numeric equivalents (particularly integer mapping), and once the computation has been performed using the neural network model, the reverse of the mapping (index to word) will be applied to visualize them. The counter from the `collections` library has been used for efficient creation of dictionaries:

```
>>> import collections
>>> counter = collections.Counter()
>>> for line in lines:
...     for word in nltk.word_tokenize(line):
...         counter[word.lower()] += 1
>>> word2idx = {w: (i+1) for i, (w, _) in enumerate(counter.most_common())}
>>> idx2word = {v: k for k, v in word2idx.items() }
```


The following code applies word-to-integer mapping and extracts the tri-grams from the embedding. Skip-gram is the methodology in which the central word is connected to both left and right adjacent words for training, and if during testing phase if it predicts correctly:

```
>>> xs = []
>>> ys = []
>>> for line in lines:
...     embedding = [word2idx[w.lower()] for w in nltk.word_tokenize(line)]
...     triples = list(nltk.trigrams(embedding))
...     w_lefts = [x[0] for x in triples]
...     w_centers = [x[1] for x in triples]
...     w_rights = [x[2] for x in triples]
...     xs.extend(w_centers)
...     ys.extend(w_lefts)
...     xs.extend(w_centers)
...     ys.extend(w_rights)
```

The following code describes that the length of the dictionary is the vocabulary size. Nonetheless, based on user specification, any custom vocabulary size can be chosen. Here, we are considering all words though!

```
>>> print (len(word2idx))
>>> vocab_size = len(word2idx)+1
```

Based on vocabulary size, all independent and dependent variables are transformed into vector representations with the following code, in which the number of rows would be the number of words and the number of columns would be the vocabulary size. The neural network model basically maps the input and output variables over the vector space:

```
>>> ohe = OneHotEncoder(n_values=vocab_size)
>>> X = ohe.fit_transform(np.array(xs).reshape(-1, 1)).todense()
>>> Y = ohe.fit_transform(np.array(ys).reshape(-1, 1)).todense()
>>> Xtrain, Xtest, Ytrain, Ytest, xstr, xsts = train_test_split(X, Y, xs,
... test_size=0.3, random_state=42)
>>> print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
```

Out of total 13,868 observations, train and test are split into 70% and 30%, which are created as 9,707 and 4,161 respectively:

```
(9707L, 1787L) (4161L, 1787L) (9707L, 1787L) (4161L, 1787L)
```

The central part of the model is described in the following few lines of deep learning code using Keras software. It is a convergent-divergent code, in which initially the dimensions of all input words are squeezed to achieve the output format.

While doing so, the dimensions are reduced to 2D in the second layer. After training the model, we will extract up to the second layer for predictions on test data. This literally works similar to the conventional encoder-decoder architecture:

```
>>> from keras.layers import Input,Dense,Dropout
>>> from keras.models import Model
>>> np.random.seed(42)
>>> BATCH_SIZE = 128
>>> NUM_EPOCHS = 20
>>> input_layer = Input(shape = (Xtrain.shape[1],),name="input")
>>> first_layer = Dense(300,activation='relu',name = "first")(input_layer)
>>> first_dropout = Dropout(0.5,name="firstdout")(first_layer)
>>> second_layer = Dense(2,activation='relu',name="second")(first_dropout)
>>> third_layer = Dense(300,activation='relu',name="third")(second_layer)
>>> third_dropout = Dropout(0.5,name="thirdout")(third_layer)
>>> fourth_layer = Dense(Ytrain.shape[1],activation='softmax',name =
"fourth")(third_dropout)
>>> history = Model(input_layer,fourth_layer)
>>> history.compile(optimizer = "rmsprop",loss= "categorical_crossentropy",
metrics=["accuracy"])
```

The following code is used to train the model:

```
>>> history.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,epochs=NUM_EPOCHS,
verbose=1,validation_split = 0.2)
```

By carefully observing the accuracy on both the training and validation datasets, we can find that the best accuracy values are not even crossing 6%. This happens due to limited data and architecture of deep learning models. In order to make this really work, we need at least gigabytes of data and large architectures. Models too need to be trained for very long. Due to practical constraints and illustration purposes, we have just trained for 20 iterations. However, readers are encouraged to try various combinations to improve the accuracy.

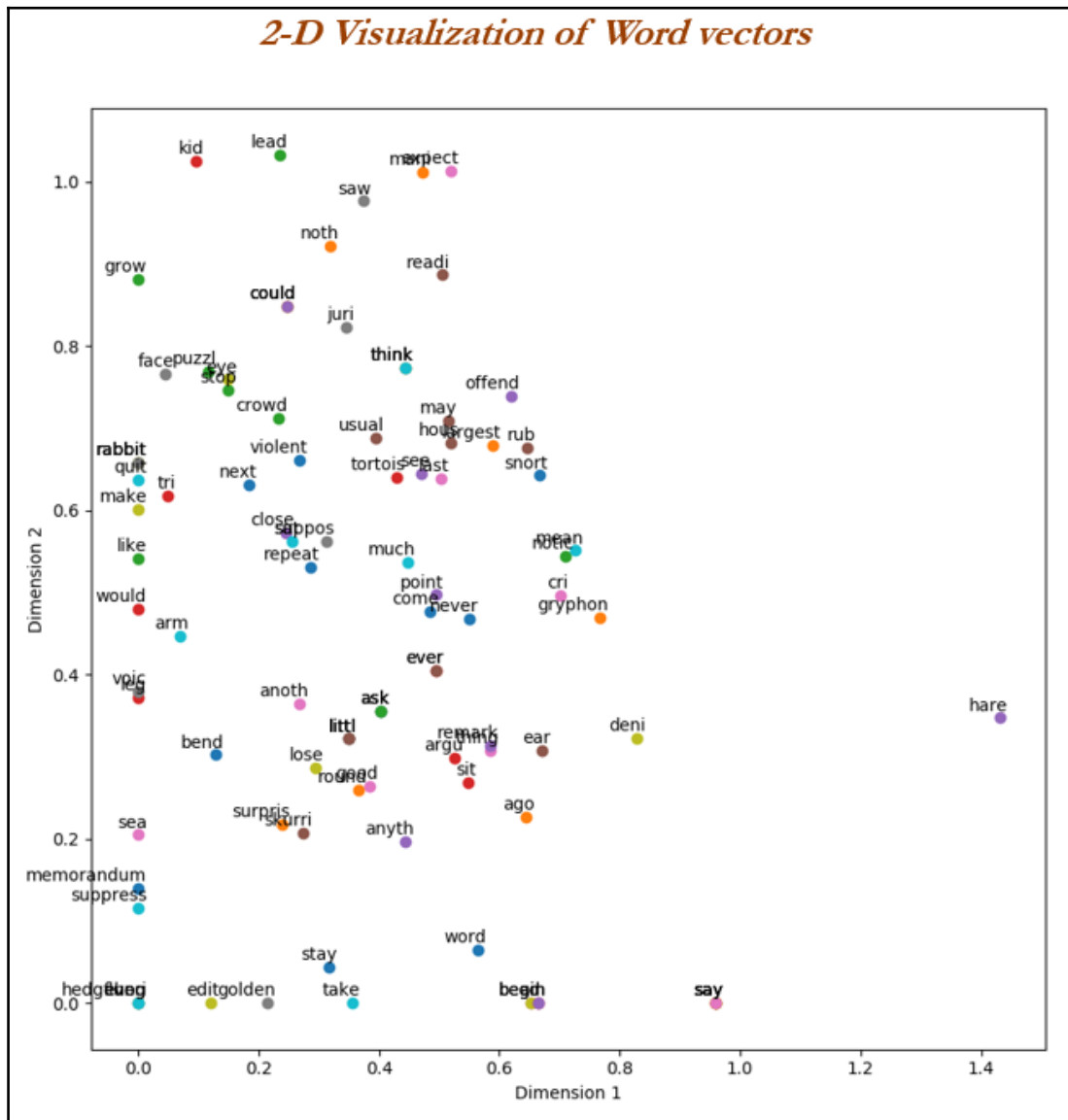
```
Train on 7765 samples, validate on 1942 samples
Epoch 1/20
7765/7765 [=====] - 0s - loss: 6.9000 - acc: 0.0382 - val_loss: 6.4369 - val_acc: 0.0479
Epoch 2/20
7765/7765 [=====] - 0s - loss: 6.3954 - acc: 0.0426 - val_loss: 6.4417 - val_acc: 0.0479
Epoch 3/20
7765/7765 [=====] - 0s - loss: 6.3458 - acc: 0.0434 - val_loss: 6.4712 - val_acc: 0.0479
Epoch 4/20
7765/7765 [=====] - 0s - loss: 6.3264 - acc: 0.0438 - val_loss: 6.4962 - val_acc: 0.0479
Epoch 5/20
7765/7765 [=====] - 0s - loss: 6.3099 - acc: 0.0439 - val_loss: 6.5160 - val_acc: 0.0479
Epoch 6/20
7765/7765 [=====] - 0s - loss: 6.2900 - acc: 0.0440 - val_loss: 6.5360 - val_acc: 0.0479
Epoch 7/20
7765/7765 [=====] - 0s - loss: 6.2700 - acc: 0.0441 - val_loss: 6.5560 - val_acc: 0.0479
Epoch 8/20
7765/7765 [=====] - 0s - loss: 6.2500 - acc: 0.0442 - val_loss: 6.5760 - val_acc: 0.0479
Epoch 9/20
7765/7765 [=====] - 0s - loss: 6.2300 - acc: 0.0443 - val_loss: 6.5960 - val_acc: 0.0479
Epoch 10/20
7765/7765 [=====] - 0s - loss: 6.2100 - acc: 0.0444 - val_loss: 6.6160 - val_acc: 0.0479
Epoch 11/20
7765/7765 [=====] - 0s - loss: 6.1900 - acc: 0.0445 - val_loss: 6.6360 - val_acc: 0.0479
Epoch 12/20
7765/7765 [=====] - 0s - loss: 6.1700 - acc: 0.0446 - val_loss: 6.6560 - val_acc: 0.0479
Epoch 13/20
7765/7765 [=====] - 0s - loss: 6.1500 - acc: 0.0447 - val_loss: 6.6760 - val_acc: 0.0479
Epoch 14/20
7765/7765 [=====] - 0s - loss: 6.1300 - acc: 0.0448 - val_loss: 6.6960 - val_acc: 0.0479
Epoch 15/20
7765/7765 [=====] - 0s - loss: 6.1100 - acc: 0.0449 - val_loss: 6.7160 - val_acc: 0.0479
Epoch 16/20
7765/7765 [=====] - 0s - loss: 6.0900 - acc: 0.0450 - val_loss: 6.7360 - val_acc: 0.0479
Epoch 17/20
7765/7765 [=====] - 0s - loss: 6.0700 - acc: 0.0451 - val_loss: 6.7560 - val_acc: 0.0479
Epoch 18/20
7765/7765 [=====] - 0s - loss: 6.0500 - acc: 0.0452 - val_loss: 6.7760 - val_acc: 0.0479
Epoch 19/20
7765/7765 [=====] - 0s - loss: 6.0300 - acc: 0.0453 - val_loss: 6.7960 - val_acc: 0.0479
Epoch 20/20
7765/7765 [=====] - 0s - loss: 6.0100 - acc: 0.0454 - val_loss: 6.8160 - val_acc: 0.0479
Out[9]: <keras.callbacks.History at 0x197984550>
```

```
# Extracting Encoder section of the Model for prediction of latent
variables
>>> encoder = Model(history.input,history.get_layer("second").output)

# Predicting latent variables with extracted Encoder model
>>> reduced_X = encoder.predict(Xtest)
Converting the outputs into Pandas data frame structure for better
representation
>>> final_pdframe = pd.DataFrame(reduced_X)
>>> final_pdframe.columns = ["xaxis","yaxis"]
>>> final_pdframe["word_indx"] = xsts
>>> final_pdframe["word"] = final_pdframe["word_indx"].map(idx2word)
>>> rows = random.sample(final_pdframe.index, 100)
>>> vis_df = final_pdframe.ix[rows]
>>> labels = list(vis_df["word"]);xvals = list(vis_df["xaxis"])
>>> yvals = list(vis_df["yaxis"])

#in inches
>>> plt.figure(figsize=(8, 8))
>>> for i, label in enumerate(labels):
...     x = xvals[i]
...     y = yvals[i]
...     plt.scatter(x, y)
...     plt.annotate(label,xy=(x, y),xytext=(5, 2),textcoords='offset points',
ha='right',va='bottom')
>>> plt.xlabel("Dimension 1")
>>> plt.ylabel("Dimension 2")
>>> plt.show()
```

The following image describes the visualization of the words in two-dimensional space. Some words are closer to each other than other words, which indicates closeness and relationships with nearby words. For example, words such as *never*, *ever*, and *ask* are very close to each other.



10

Advanced Applications of Deep Learning in NLP

In this chapter, we will cover the following advanced recipes:

- Automated text generation from Shakespeare's writings using LSTM
- Questions and answers on episodic data using memory networks
- Language modeling to predict the next best word using recurrent neural networks – LSTM
- Generative chatbot development using deep learning recurrent networks – LSTM

Introduction

Deep learning techniques are being utilized well to solve some open-ended problems. This chapter discusses these types of problems, in which a simple *yes* or *no* would be difficult to say. We are hopeful that you will enjoy going through these recipes to obtain the viewpoint of what cutting-edge works are going on in this industry at the moment, and try to learn some basic building blocks of the same with relevant coding snippets.

Automated text generation from Shakespeare's writings using LSTM

In this recipe, we will use deep **recurrent neural networks** (RNN) to predict the next character based on the given length of a sentence. This way of training a model can generate automated text continuously, which imitates the writing style of the original writer with enough training on the number of epochs and so on.

Getting ready...

The *Project Gutenberg* eBook of the complete works of William Shakespeare's dataset is used to train the network for automated text generation. Data can be downloaded from <http://www.gutenberg.org/> for the raw file used for training:

```
>>> from __future__ import print_function
>>> import numpy as np
>>> import random
>>> import sys
```

The following code is used to create a dictionary of characters to indices and vice-versa mapping, which we will be using to convert text into indices at later stages. This is because deep learning models cannot understand English and everything needs to be mapped into indices to train these models:

```
>>> path = 'C:\\Users\\prata\\Documents\\book_codes\\ NLP_DL\\
shakespeare_final.txt'
>>> text = open(path).read().lower()
>>> characters = sorted(list(set(text)))
>>> print('corpus length:', len(text))
>>> print('total chars:', len(characters))
```

```
corpus length: 581432
total chars: 61
```

```
>>> char2indices = dict((c, i) for i, c in enumerate(characters))
>>> indices2char = dict((i, c) for i, c in enumerate(characters))
```

How to do it...

Before training the model, various preprocessing steps are involved to make it work. The following are the major steps involved:

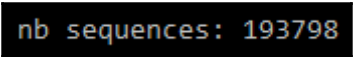
1. **Preprocessing:** Prepare X and Y data from the given entire story text file and converting them into indices vectorized format.
2. **Deep learning model training and validation:** Train and validate the deep learning model.
3. **Text generation:** Generate the text with the trained model.

How it works...

The following lines of code describe the entire modeling process of generating text from Shakespeare's writings. Here we have chosen character length. This needs to be considered as 40 to determine the next best single character, which seems to be very fair to consider. Also, this extraction process jumps by three steps to avoid any overlapping between two consecutive extractions, to create a dataset more fairly:

```
# cut the text in semi-redundant sequences of maxlen characters
>>> maxlen = 40
>>> step = 3
>>> sentences = []
>>> next_chars = []
>>> for i in range(0, len(text) - maxlen, step):
...     sentences.append(text[i: i + maxlen])
...     next_chars.append(text[i + maxlen])
...     print('nb sequences:', len(sentences))
```

The following screenshot depicts the total number of sentences considered, 193798, which is enough data for text generation:



```
nb sequences: 193798
```

The next code block is used to convert the data into a vectorized format for feeding into deep learning models, as the models cannot understand anything about text, words, sentences and so on. Initially, total dimensions are created with all zeros in the NumPy array and filled with relevant places with dictionary mappings:

```
# Converting indices into vectorized format
>>> X = np.zeros((len(sentences), maxlen, len(characters)), dtype=np.bool)
>>> y = np.zeros((len(sentences), len(characters)), dtype=np.bool)
```

```
>>> for i, sentence in enumerate(sentences):
...     for t, char in enumerate(sentence):
...         X[i, t, char2indices[char]] = 1
...         y[i, char2indices[next_chars[i]]] = 1
>>> from keras.models import Sequential
>>> from keras.layers import Dense, LSTM, Activation, Dropout
>>> from keras.optimizers import RMSprop
```

The deep learning model is created with RNN, more specifically Long Short-Term Memory networks with 128 hidden neurons, and the output is in the dimensions of the characters. The number of columns in the array is the number of characters. Finally, the `softmax` function is used with the `RMSprop` optimizer. We encourage readers to try with other various parameters to check out how results vary:

```
#Model Building
>>> model = Sequential()
>>> model.add(LSTM(128, input_shape=(maxlen, len(characters))))
>>> model.add(Dense(len(characters)))
>>> model.add(Activation('softmax'))
>>> model.compile(loss='categorical_crossentropy',
optimizer=RMSprop(lr=0.01))
>>> print (model.summary())
```

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 128)	97280
dense_1 (Dense)	(None, 61)	7869
activation_1 (Activation)	(None, 61)	0
Total params: 105,149.0		
Trainable params: 105,149.0		
Non-trainable params: 0.0		

As mentioned earlier, deep learning models train on number indices to map input to output (given a length of 40 characters, the model will predict the next best character). The following code is used to convert the predicted indices back to the relevant character by determining the maximum index of the character:

```
# Function to convert prediction into index
>>> def pred_indices(preds, metric=1.0):
...     preds = np.asarray(preds).astype('float64')
...     preds = np.log(preds) / metric
...     exp_preds = np.exp(preds)
```

```
... preds = exp_preds/np.sum(exp_preds)
... probs = np.random.multinomial(1, preds, 1)
... return np.argmax(probs)
```

The model will be trained over 30 iterations with a batch size of 128. And also, the diversity has been changed to see the impact on the predictions:

```
# Train and Evaluate the Model
>>> for iteration in range(1, 30):
...     print('-' * 40)
...     print('Iteration', iteration)
...     model.fit(X, y, batch_size=128, epochs=1)
...     start_index = random.randint(0, len(text) - maxlen - 1)
...     for diversity in [0.2, 0.7, 1.2]:
...         print('\n----- diversity:', diversity)
...         generated = ''
...         sentence = text[start_index: start_index + maxlen]
...         generated += sentence
...         print('----- Generating with seed: "' + sentence + '"')
...         sys.stdout.write(generated)
...         for i in range(400):
...             x = np.zeros((1, maxlen, len(characters)))
...             for t, char in enumerate(sentence):
...                 x[0, t, char2indices[char]] = 1.
...             preds = model.predict(x, verbose=0)[0]
...             next_index = pred_indices(preds, diversity)
...             pred_char = indices2char[next_index]
...             generated += pred_char
...             sentence = sentence[1:] + pred_char
...             sys.stdout.write(pred_char)
...             sys.stdout.flush()
...         print("\nOne combination completed \n")
```

The results are shown in the next screenshot to compare the first iteration (Iteration 1) and final iteration (Iteration 29). It is apparent that with enough training, the text generation seems to be much better than with Iteration 1:

```

-----
Iteration 1
Epoch 1/1
193798/193798 [=====] - 77s - loss: 1.8861

----- diversity: 0.2
----- Generating with seed: "palace

enter cleopatra, charmian, iras,"
palace

enter cleopatra, charmian, iras, a menter i will so a more a more have hour to more here so
more to the seell was to the some
    when the some hour the some of even a more heart here will some have shall me the some w
orth,
    a more horse worth, a more horse to be the comments
    a to the some a more the soldier to the part hour the love,
    i a
combination completed
One

----- diversity: 0.7
----- Generating with seed: "palace

enter cleopatra, charmian, iras,"
palace

enter cleopatra, charmian, iras, a for day
    for more honiudes of hone what stonder i come him;
    but whot not my eart a foulfiver my so my bro her. most a mest rost must vingured soldier,

        axit helster belowe i will out a fortuneess.
    porsest evenen dast you will not our forther but belong to that hear i mongherOne combinat
ion completed

----- diversity: 1.2
----- Generating with seed: "palace

enter cleopatra, charmian, iras,"
palace

enter cleopatra, charmian, iras, why.
i draboutur in misherss so brintow upon
    you bros! to that hath womant hons; and tweereph brountlally,
    and appercy, he your lork! beweiv a lis bitelicr,
        ohony now fortime, pome
sake know he have whenchups, you gos, xous
    ifor mitht a
        ofverguss love?' hishwadsh, youra year trougt soumshorneks, and lewas! be soleteites-
soldients, thinksty

```

Text generation after Iteration 29 is shown in this image:

```

-----
Iteration 29
Epoch 1/1
193798/193798 [=====] - 84s - loss: 1.2596

---- diversity: 0.2
---- Generating with seed: "to beneath your constable, it will fit
"
to beneath your constable, it will fit
    and make the death and seem and she such counterfors,
    and i am steer place and the such so many
    the strength the way the beautions and man,
    i will not the world so straight to see him.
    what shall see thy faith the world and for my heart.
    i am so well my self with antony, and is
provided by the fortunes and madam.
    cleopatra. a prove the world with the world see the fortunesOne combination completed

---- diversity: 0.7
---- Generating with seed: "to beneath your constable, it will fit
"
to beneath your constable, it will fit
    that we'll see him, that must thou in my excrueds,
    good of my surely slow gone to seek she grace
    of my unwore the fortunes with when what shall not out of that acceptain's war.
    but i have ages the strength such strange
    but an these fortunes, and still i do hearday shall see have you love to let us.
    your self in stedents and great the things not do friends is that be will
    One combination completed

---- diversity: 1.2
---- Generating with seed: "to beneath your constable, it will fit
"
to beneath your constable, it will fit
    untled root silts cleopatra!
    ty haths and furethance of gentle, 'teverald once
    firthers.
    woe no, my age gait, marry of nature'd,
    and from duke ower cannot came on out yout.
    atteun! dies, sir. [asidit! gender let thie, a love'
    . no own mal, writh any love.
    alexas. this it hath abown of till the twongian madies.
pelo
    behaesa of well here,
    a get thee with past errike; speaking aOne combination completed

```

Though the text generation altogether seems to be a bit magical, we have generated text using Shakespeare's writings, proving that with the right training and handling, we can imitate any writer's style of writing.

Questions and answers on episodic data using memory networks

In this recipe, we will use deep RNN to create a model to work on a question-and-answer system based on episodic memory. It will extract the relevant answers for a given question by reading a story in a sequential manner. For further reading, refer to the paper *Dynamic Memory Networks for Natural Language Processing* by Ankit Kumar et. al. (<https://arxiv.org/pdf/1506.07285.pdf>).

Getting ready...

Facebook's bAbI data has been used for this example, and the same can be downloaded from http://www.thespermwhale.com/jaseweston/babi/tasks_1-20_v1-2.tar.gz. It consists of about 20 types of tasks, among which we have taken the first one, a single supporting-fact-based question-and-answer system.

After unzipping the file, go to the en-10k folder and use the files starting with qa1_single supporting-fact for both the train and test files. The following code is used for extraction of stories, questions, and answers in this particular order to create the data required for training:

```
>>> from __future__ import division, print_function
>>> import collections
>>> import itertools
>>> import nltk
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import os
>>> import random
>>> def get_data(infile):
...     stories, questions, answers = [], [], []
...     story_text = []
...     fin = open(Train_File, "rb")
...     for line in fin:
...         line = line.decode("utf-8").strip()
...         lno, text = line.split(" ", 1)
...         if "\t" in text:
```

```
... question, answer, _ = text.split("\t")
... stories.append(story_text)
... questions.append(question)
... answers.append(answer)
... story_text = []
... else:
...     story_text.append(text)
... fin.close()
... return stories, questions, answers
>>> file_location = "C:/Users/prata/Documents/book_codes/NLP_DL"
>>> Train_File = os.path.join(file_location, "qa1_single-supporting-
fact_train.txt")
>>> Test_File = os.path.join(file_location, "qa1_single-supporting-
fact_test.txt")
# get the data
>>> data_train = get_data(Train_File)
>>> data_test = get_data(Test_File)
>>> print("\n\nTrain observations:", len(data_train[0]), "Test
observations:", len(data_test[0]), "\n\n")
```

After extraction, it seems that about 10k observations were created in the data for both train and test datasets:

```
Train observations: 10000 Test observations: 10000
```

How to do it...

After extraction of basic datasets, we need to follow these steps:

1. **Preprocessing:** Create a dictionary and map the story, question and answers to vocab to map into vector format.
2. **Model development and validation:** Train the deep learning models and test on the validation data sample.
3. **Predicting outcomes based on the trained model:** Trained models are utilized for predicting outcomes on test data.

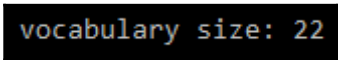
How it works...

After train and test data creation, the remaining methodology is described as follows.

First, we will create a dictionary for vocabulary, in which for every word from the story, question and answer data mapping is created. Mappings are used to convert words into integer numbers and subsequently into vector space:

```
# Building Vocab dictionary from Train and Test data
>>> dictnry = collections.Counter()
>>> for stories, questions, answers in [data_train, data_test]:
...     for story in stories:
...         for sent in story:
...             for word in nltk.word_tokenize(sent):
...                 dictnry[word.lower()] +=1
...     for question in questions:
...         for word in nltk.word_tokenize(question):
...             dictnry[word.lower()] +=1
...     for answer in answers:
...         for word in nltk.word_tokenize(answer):
...             dictnry[word.lower()] +=1
>>> word2indx = {w: (i+1) for i, (w, _) in enumerate(dictnry.most_common())}
>>> word2indx["PAD"] = 0
>>> indx2word = {v: k for k, v in word2indx.items()}
>>> vocab_size = len(word2indx)
>>> print("vocabulary size:", len(word2indx))
```

The following screenshot depicts all the words in the vocabulary. It has only 22 words, including the PAD word, which has been created to fill blank spaces or zeros:



```
vocabulary size: 22
```

The following code is used to determine the maximum length of words. By knowing this, we can create a vector of maximum size, which can incorporate all lengths of words:

```
# compute max sequence length for each entity
>>> story_maxlen = 0
>>> question_maxlen = 0
>>> for stories, questions, answers in [data_train, data_test]:
...     for story in stories:
...         story_len = 0
...         for sent in story:
...             swords = nltk.word_tokenize(sent)
...             story_len += len(swords)
...         if story_len > story_maxlen:
...             story_maxlen = story_len
```

```
... for question in questions:
...     question_len = len(nltk.word_tokenize(question))
...     if question_len > question_maxlen:
...         question_maxlen = question_len>>> print ("Story maximum
length:",story_maxlen,"Question maximum length:",question_maxlen)
```

The maximum length of words for story is 14, and for questions it is 4. For some of the stories and questions, the length could be less than the maximum length; those words will be replaced with 0 (or PAD word). The reason? This padding of extra blanks will make all the observations of equal length. This is for computation efficiency, or else it will be difficult to map different lengths, or creating parallelization in GPU for computation will be impossible:

```
Story maximum length: 14 Question maximum length: 4
```

Following snippets of code does import various functions from respective classes which we will be using in the following section:

```
>>> from keras.layers import Input
>>> from keras.layers.core import Activation, Dense, Dropout, Permute
>>> from keras.layers.embeddings import Embedding
>>> from keras.layers.merge import add, concatenate, dot
>>> from keras.layers.recurrent import LSTM
>>> from keras.models import Model
>>> from keras.preprocessing.sequence import pad_sequences
>>> from keras.utils import np_utils
```

Word-to-vectorized mapping is being performed in the following code after considering the maximum lengths for story, question, and so on, while also considering vocab size, all of which we have computed in the preceding segment of code:

```
# Converting data into Vectorized form
>>> def data_vectorization(data, word2indx, story_maxlen, question_maxlen):
...     Xs, Xq, Y = [], [], []
...     stories, questions, answers = data
...     for story, question, answer in zip(stories, questions, answers):
...         xs = [word2indx[w.lower()] for w in nltk.word_tokenize(s)]
...         for s in story:
...             xs = list(itertools.chain.from_iterable(xs))
...             xq = [word2indx[w.lower()] for w in nltk.word_tokenize (question)]
...             Xs.append(xs)
...             Xq.append(xq)
...             Y.append(word2indx[answer.lower()])
...     return pad_sequences(Xs, maxlen=story_maxlen), pad_sequences(Xq,
maxlen=question_maxlen), np_utils.to_categorical(Y, num_classes=
len(word2indx))
```

The application of `data_vectorization` is shown in this code:

```
>>> Xstrain, Xqtrain, Ytrain = data_vectorization(data_train, word2indx,
story_maxlen, question_maxlen)
>>> Xstest, Xqtest, Ytest = data_vectorization(data_test, word2indx,
story_maxlen, question_maxlen)
>>> print("Train story",Xstrain.shape,"Train question",
Xqtrain.shape,"Train answer", Ytrain.shape)
>>> print( "Test story",Xstest.shape, "Test question",Xqtest.shape, "Test
answer",Ytest.shape)
```

The following image describes the dimensions of train and test data for story, question, and answer segments accordingly:

```
Train story (10000L, 14L) Train question (10000L, 4L) Train answer (10000L, 22L)
Test story (10000L, 14L) Test question (10000L, 4L) Test answer (10000L, 22L)
```

Parameters are initialized in the following code:

```
# Model Parameters
>>> EMBEDDING_SIZE = 128
>>> LATENT_SIZE = 64
>>> BATCH_SIZE = 64
>>> NUM_EPOCHS = 40
```

The core building blocks of the model are explained here:

```
# Inputs
>>> story_input = Input(shape=(story_maxlen,))
>>> question_input = Input(shape=(question_maxlen,))

# Story encoder embedding
>>> story_encoder = Embedding(input_dim=vocab_size,
output_dim=EMBEDDING_SIZE,input_length= story_maxlen) (story_input)
>>> story_encoder = Dropout(0.2) (story_encoder)

# Question encoder embedding
>>> question_encoder = Embedding(input_dim=vocab_size,output_dim=
EMBEDDING_SIZE, input_length=question_maxlen) (question_input)
>>> question_encoder = Dropout(0.3) (question_encoder)

# Match between story and question
>>> match = dot([story_encoder, question_encoder], axes=[2, 2])

# Encode story into vector space of question
>>> story_encoder_c = Embedding(input_dim=vocab_size,
output_dim=question_maxlen,input_length= story_maxlen) (story_input)
```



```

>>> story_encoder_c = Dropout(0.3)(story_encoder_c)

# Combine match and story vectors
>>> response = add([match, story_encoder_c])
>>> response = Permute((2, 1))(response)

# Combine response and question vectors to answers space
>>> answer = concatenate([response, question_encoder], axis=-1)
>>> answer = LSTM(LATENT_SIZE)(answer)
>>> answer = Dropout(0.2)(answer)
>>> answer = Dense(vocab_size)(answer)
>>> output = Activation("softmax")(answer)
>>> model = Model(inputs=[story_input, question_input], outputs=output)
>>> model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
>>> print(model.summary())

```

By reading the model summary in following image, you can see how blocks are connected and the see total number of parameters required to be trained to tune the model:

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 14)	0	
input_2 (InputLayer)	(None, 4)	0	
embedding_1 (Embedding)	(None, 14, 128)	2816	input_1
embedding_2 (Embedding)	(None, 4, 128)	2816	input_2
dropout_1 (Dropout)	(None, 14, 128)	0	embedding_1
dropout_2 (Dropout)	(None, 4, 128)	0	embedding_2
embedding_3 (Embedding)	(None, 14, 4)	88	dropout_1
dot_1 (Dot)	(None, 14, 4)	0	dropout_2
dropout_3 (Dropout)	(None, 14, 4)	0	dot_1
add_1 (Add)	(None, 14, 4)	0	dropout_3
permute_1 (Permute)	(None, 4, 14)	0	add_1
concatenate_1 (Concatenate)	(None, 4, 142)	0	permute_1
lstm_1 (LSTM)	(None, 64)	52992	concatenate_1
dropout_4 (Dropout)	(None, 64)	0	lstm_1
dense_1 (Dense)	(None, 22)	1430	dropout_4
activation_1 (Activation)	(None, 22)	0	dense_1
Total params: 60,142.0			
Trainable params: 60,142.0			
Non-trainable params: 0.0			

Following code does perform model fitting on train data:

```
# Model Training
>>> history = model.fit([Xstrain, Xqtrain], [Ytrain],
    batch_size=BATCH_SIZE, epochs=NUM_EPOCHS, validation_data= ([Xstest, Xqtest],
    [Ytest]))
```

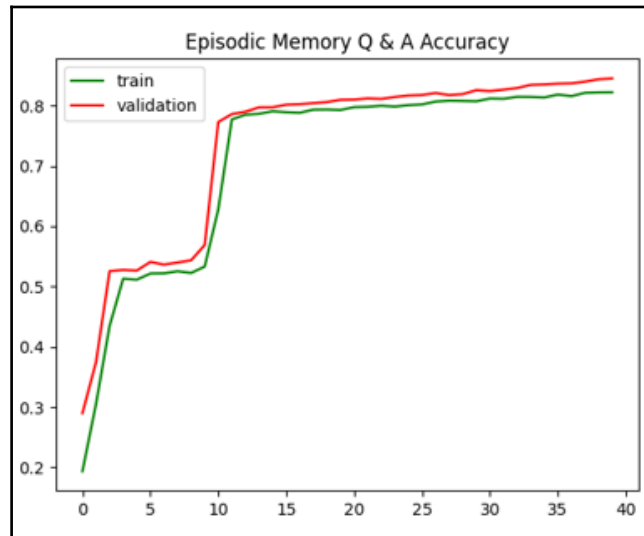
The model accuracy has significantly improved from the first iteration (*train accuracy* = 19.35% and *validation accuracy* = 28.98%) to the 40th (*train accuracy* = 82.22% and *validation accuracy* = 84.51%), which can be shown in the following image:

```
Train on 10000 samples, validate on 10000 samples
Epoch 1/40
10000/10000 [=====] - 2s - loss: 2.0210 - acc: 0.1935 - val_loss: 1.6487 - val_acc: 0.2898
Epoch 2/40
10000/10000 [=====] - 2s - loss: 1.6443 - acc: 0.3052 - val_loss: 1.5560 - val_acc: 0.3746
Epoch 3/40
10000/10000 [=====] - 2s - loss: 1.5162 - acc: 0.4347 - val_loss: 1.3883 - val_acc: 0.5255
Epoch 4/40
10000/10000 [=====] - 2s - loss: 1.3762 - acc: 0.5130 - val_loss: 1.2920 - val_acc: 0.5274
Epoch 5/40
10000/10000 [=====] - 2s - loss: 1.3155 - acc: 0.5113 - val_loss: 1.2540 - val_acc: 0.5264
Epoch 6/40
10000/10000 [=====] - 2s - loss: 1.2806 - acc: 0.5215 - val_loss: 1.2302 - val_acc: 0.5408
Epoch 36/40
10000/10000 [=====] - 2s - loss: 0.4343 - acc: 0.8184 - val_loss: 0.4126 - val_acc: 0.8365
Epoch 37/40
10000/10000 [=====] - 2s - loss: 0.4357 - acc: 0.8158 - val_loss: 0.4104 - val_acc: 0.8370
Epoch 38/40
10000/10000 [=====] - 2s - loss: 0.4297 - acc: 0.8213 - val_loss: 0.4070 - val_acc: 0.8399
Epoch 39/40
10000/10000 [=====] - 2s - loss: 0.4311 - acc: 0.8220 - val_loss: 0.4053 - val_acc: 0.8438
Epoch 40/40
10000/10000 [=====] - 2s - loss: 0.4260 - acc: 0.8222 - val_loss: 0.4015 - val_acc: 0.8451
```

Following code does plot both training & validation accuracy change with respective to change in epoch:

```
# plot accuracy and loss plot
>>> plt.title("Accuracy")
>>> plt.plot(history.history["acc"], color="g", label="train")
>>> plt.plot(history.history["val_acc"], color="r", label="validation")
>>> plt.legend(loc="best")
>>> plt.show()
```

The change in accuracy with the number of iterations is shown in the following image. It seems that the accuracy has improved marginally rather than drastically after 10 iterations:



In the following code, results are predicted which is finding probability for each respective class and also applying `argmax` function for finding the class where the probability is maximum:

```
# get predictions of labels
>>> ytest = np.argmax(Ytest, axis=1)
>>> Ytest_ = model.predict([Xstest, Xqtest])
>>> ytest_ = np.argmax(Ytest_, axis=1)
# Select Random questions and predict answers
>>> NUM_DISPLAY = 10
>>> for i in random.sample(range(Xstest.shape[0]), NUM_DISPLAY):
...     story = " ".join([indx2word[x] for x in Xstest[i].tolist() if x != 0])
...     question = " ".join([indx2word[x] for x in Xqtest[i].tolist()])
...     label = indx2word[ytest[i]]
...     prediction = indx2word[ytest_[i]]
...     print(story, question, label, prediction)
```

After training the model enough and achieving better accuracies on validation data such as 84.51%, it is time to verify with actual test data to see how much the predicted answers are in line with the actual answers.

Out of ten randomly drawn questions, the model was unable to predict the correct question only once (for the sixth question; the actual answer is `bedroom` and the predicted answer is `office`). This means we have got 90% accuracy on the sample. Though we may not be able to generalize the accuracy value, this gives some idea to reader about the prediction ability of the model:

```
mary journeyed to the kitchen . daniel went to the bedroom . where is daniel ? bedroom bedroom
daniel went back to the hallway . sandra went to the garden . where is sandra ? garden garden
sandra journeyed to the hallway . sandra journeyed to the bathroom . where is sandra ? bathroom bathroom
john travelled to the bedroom . daniel moved to the garden . where is daniel ? garden garden
sandra journeyed to the hallway . sandra travelled to the kitchen . where is mary ? bathroom bathroom
daniel moved to the bathroom . daniel journeyed to the hallway . where is john ? bedroom office
john went to the hallway . daniel travelled to the hallway . where is daniel ? hallway hallway
john went back to the bathroom . sandra went back to the hallway . where is sandra ? hallway hallway
john went to the office . john went to the bedroom . where is john ? bedroom bedroom
john travelled to the kitchen . daniel travelled to the bathroom . where is daniel ? bathroom bathroom
```

Language modeling to predict the next best word using recurrent neural networks LSTM

Predicting the next word based on some typed words has many real-world applications. An example would be to suggest the word while typing it into the Google search bar. This type of feature does improve user satisfaction in using search engines. Technically, this can be called **N-grams** (if two consecutive words are extracted, it will be called **bi-grams**). Though there are so many ways to model this, here we have chosen deep RNNs to predict the next best word based on *N-1* pre-words.

Getting ready...

Alice in Wonderland data has been used for this purpose and the same data can be downloaded from <http://www.umich.edu/~umfandsf/other/ebooks/alice30.txt>. In the initial data preparation stage, we have extracted N-grams from continuous text file data, which looks like a story file:

```
>>> from __future__ import print_function
>>> import os
""" First change the following directory link to where all input files do
exist """
>>> os.chdir("C:\\Users\\prata\\Documents\\book_codes\\NLP_DL")
>>> from sklearn.model_selection import train_test_split
>>> import nltk
>>> import numpy as np
>>> import string
```

```
# File reading
>>> with open('alice_in_wonderland.txt', 'r') as content_file:
...     content = content_file.read()
>>> content2 = " ".join("".join([" " if ch in string.punctuation else ch
for ch in content])).split()
>>> tokens = nltk.word_tokenize(content2)
>>> tokens = [word.lower() for word in tokens if len(word)>=2]
```

N-grams are selected with the following N value. In the following code, we have chosen N as 3, which means each piece of data has three words consecutively. Among them, two pre-words (bi-grams) used to predict the next word in each observation. Readers are encouraged to change the value of N and see how the model predicts the words:



Note: With the increase in N-grams to 4, 5, and 6 or so, we need to provide enough amount of incremental data to compensate for the curse of dimensionality.

```
# Select value of N for N grams among which N-1 are used to predict last
Nth word
>>> N = 3
>>> quads = list(nltk.ngrams(tokens, N))
>>> newl_app = []
>>> for ln in quads:
...     newl = " ".join(ln)
...     newl_app.append(newl)
```

How to do it...

After extracting basic data observations, we need to perform the following operations:

1. **Preprocessing:** In the preprocessing step, words are converted to vectorized form, which is needed for working with the model.
2. **Model development and validation:** Create a convergent-divergent model to map the input to the output, followed by training and validation data.
3. **Prediction of next best word:** Utilize the trained model to predict the next best word on test data.

How it works...

Vectorization of the given words (X and Y words) to vector space using CountVectorizer from scikit-learn:

```
# Vectorizing the words
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer()
>>> x_trigm = []
>>> y_trigm = []
>>> for l in newl_app:
...     x_str = " ".join(l.split()[0:N-1])
...     y_str = l.split()[N-1]
...     x_trigm.append(x_str)
...     y_trigm.append(y_str)
>>> x_trigm_check = vectorizer.fit_transform(x_trigm).todense()
>>> y_trigm_check = vectorizer.fit_transform(y_trigm).todense()
# Dictionaries from word to integer and integer to word
>>> dictnry = vectorizer.vocabulary_
>>> rev_dictnry = {v:k for k,v in dictnry.items()}
>>> X = np.array(x_trigm_check)
>>> Y = np.array(y_trigm_check)
>>> Xtrain, Xtest, Ytrain, Ytest, xtrain_tg, xtest_tg = train_test_split(X,
Y, x_trigm, test_size=0.3, random_state=42)
>>> print("X Train shape", Xtrain.shape, "Y Train shape" , Ytrain.shape)
>>> print("X Test shape", Xtest.shape, "Y Test shape" , Ytest.shape)
```

After converting the data into vectorized form, we can see that the column value remains the same, which is the vocabulary length (2559 of all possible words):

```
X Train shape (17947L, 2559L) Y Train shape (17947L, 2559L)
X Test shape (7692L, 2559L) Y Test shape (7692L, 2559L)
```

The following code is the heart of the model, consisting of convergent-divergent architecture that reduces and expands the shape of the neural network:

```
# Model Building
>>> from keras.layers import Input, Dense, Dropout
>>> from keras.models import Model
>>> np.random.seed(42)
>>> BATCH_SIZE = 128
>>> NUM_EPOCHS = 100
>>> input_layer = Input(shape = (Xtrain.shape[1],), name="input")
>>> first_layer = Dense(1000, activation='relu', name = "first")(input_layer)
>>> first_dropout = Dropout(0.5, name="firstdout")(first_layer)
>>> second_layer = Dense(800, activation='relu', name="second")
```

```

(first_dropout)
>>> third_layer = Dense(1000,activation='relu',name="third") (second_layer)
>>> third_dropout = Dropout(0.5,name="thirdout") (third_layer)
>>> fourth_layer = Dense(Ytrain.shape[1],activation='softmax',name =
"fourth") (third_dropout)
>>> history = Model(input_layer,fourth_layer)
>>> history.compile(optimizer = "adam",loss="categorical_crossentropy",
metrics=["accuracy"])
>>> print (history.summary())

```

This screenshot depicts the complete architecture of the model, consisting of a convergent-divergent structure:

Layer (type)	Output Shape	Param #
input (InputLayer)	(None, 2559L)	0
first (Dense)	(None, 1000)	2560000
firstdout (Dropout)	(None, 1000)	0
second (Dense)	(None, 800)	800800
third (Dense)	(None, 1000)	801000
thirdout (Dropout)	(None, 1000)	0
fourth (Dense)	(None, 2559L)	2561559
Total params: 6,723,359.0		
Trainable params: 6,723,359.0		
Non-trainable params: 0.0		

```

# Model Training
>>> history.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,epochs=NUM_EPOCHS,
verbose=1,validation_split = 0.2)

```

The model is trained on data with 100 epochs. Even after a significant improvement in the train accuracy (from 5.46% to 63.18%), there is little improvement in the validation accuracy (6.63% to 10.53%). However, readers are encouraged to try various settings to improve the validation accuracy further:

```
Train on 14357 samples, validate on 3590 samples
Epoch 1/100
14357/14357 [=====] - 1s - loss: 6.3349 - acc: 0.0546 - val_loss: 6.0973 - val_acc: 0.0663
Epoch 2/100
14357/14357 [=====] - 1s - loss: 5.9002 - acc: 0.0664 - val_loss: 6.0327 - val_acc: 0.0733
Epoch 3/100
14357/14357 [=====] - 1s - loss: 5.6806 - acc: 0.0823 - val_loss: 5.9812 - val_acc: 0.0869
Epoch 4/100
14357/14357 [=====] - 1s - loss: 5.4250 - acc: 0.1045 - val_loss: 5.9641 - val_acc: 0.0969
Epoch 96/100
14357/14357 [=====] - 1s - loss: 1.1159 - acc: 0.6394 - val_loss: 9.2412 - val_acc: 0.1100
Epoch 97/100
14357/14357 [=====] - 1s - loss: 1.1252 - acc: 0.6329 - val_loss: 9.2342 - val_acc: 0.1100
Epoch 98/100
14357/14357 [=====] - 1s - loss: 1.1061 - acc: 0.6375 - val_loss: 9.3985 - val_acc: 0.1120
Epoch 99/100
14357/14357 [=====] - 1s - loss: 1.1132 - acc: 0.6368 - val_loss: 9.3619 - val_acc: 0.1092
Epoch 100/100
14357/14357 [=====] - 1s - loss: 1.1138 - acc: 0.6318 - val_loss: 9.2746 - val_acc: 0.1053
Out[11]: <keras.callbacks.History at 0xe16dc4f98>
```

```
# Model Prediction
>>> Y_pred = history.predict(Xtest)
# Sample check on Test data
>>> print ("Prior bigram words", "|Actual", "|Predicted", "\n")
>>> for i in range(10):
...     print (i, xtest_tg[i], "|", rev_dictnry[np.argmax(Ytest[i])],
...           "|", rev_dictnry[np.argmax(Y_pred[i])])
```

Less validation accuracy provides a hint that the model might not predict the word very well. The reason could be the very-high-dimensional aspect of taking the word rather than the character level (character dimensions are 26, which is much less than the 2559 value of words). In the following screenshot, we have predicted about two times out of 10. However, it is very subjective to say whether it is a yes or no. Sometimes, the word predicted could be close but not the same:

```
7148 want to | go | see
3039 neck nicely | straightened | of
2408 the rest | between | of
4068 soon finished | off | it
7093 up and | began | down
6885 her so | she | she
5985 was an | old | old
4901 have been | that | changed
4447 the roof | there | of
777 no lower | said | the
```


Generative chatbot using recurrent neural networks (LSTM)

Generative chatbots are very difficult to build and operate. Even today, most workable chatbots are retrieving in nature; they retrieve the best response for the given question based on semantic similarity, intent, and so on. For further reading, refer to the paper *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation* by Kyunghyun Cho et. al. (<https://arxiv.org/pdf/1406.1078.pdf>).

Getting ready...

The A.L.I.C.E Artificial Intelligence Foundation dataset `bot.aiml` **Artificial Intelligence Markup Language (AIML)**, which is customized syntax such as XML file has been used to train the model. In this file, questions and answers are mapped. For each question, there is a particular answer. Complete `.aiml` files are available at *aiml-en-us-foundation-alice.v1-9* from <https://code.google.com/archive/p/aiml-en-us-foundation-alice/downloads>. Unzip the folder to see the `bot.aiml` file and open it using Notepad. Save as `bot.txt` to read in Python:

```
>>> import os
""" First change the following directory link to where all input files do
exist """
>>> os.chdir("C:\\Users\\prata\\Documents\\book_codes\\NLP_DL")
>>> import numpy as np
>>> import pandas as pd
# File reading
>>> with open('bot.txt', 'r') as content_file:
... botdata = content_file.read()
>>> Questions = []
>>> Answers = []
```

AIML files have unique syntax, similar to XML. The `pattern` word is used to represent the question and the `template` word for the answer. Hence, we are extracting respectively:

```
>>> for line in botdata.split("</pattern>"):
... if "<pattern>" in line:
... Quesn = line[line.find("<pattern>")+len("<pattern>"):]
... Questions.append(Quesn.lower())
>>> for line in botdata.split("</template>"):
... if "<template>" in line:
... Ans = line[line.find("<template>")+len("<template>"):]
... Ans = Ans.lower()
... Answers.append(Ans.lower())
```

```
>>> QnAdata = pd.DataFrame(np.column_stack([Questions, Answers]), columns =
["Questions", "Answers"])
>>> QnAdata["QnAcomb"] = QnAdata["Questions"]+" "+QnAdata["Answers"]
>>> print(QnAdata.head())
```

The question and answers are joined to extract the total vocabulary used in the modeling, as we need to convert all words/characters into numeric representation. The reason is the same as mentioned before—deep learning models can't read English and everything is in numbers for the model.

```

      Questions                               Answers \
0         yahoo  a lot of people hear about <bot name="name"/> ...
1      you are lazy                actually i work 24 hours a day.
2      you are mad                no i am quite logical and rational.
3  you are thinking  i am a thinking machine.<think><set name="it">...
4  you are dividing *          actually i am not too good at division.

      QnAcomb
0  yahoo a lot of people hear about <bot name="na...
1      you are lazy actually i work 24 hours a day.
2      you are mad no i am quite logical and rational.
3  you are thinking i am a thinking machine.<thin...
4  you are dividing * actually i am not too good ...

```

How to do it...

After extracting the question-and-answer pairs, the following steps are needed to process the data and produce the results:

1. **Preprocessing:** Convert the question-and-answer pairs into vectorized format, which will be utilized in model training.
2. **Model building and validation:** Develop deep learning models and validate the data.
3. **Prediction of answers from trained model:** The trained model will be used to predict answers for given questions.

How it works...

The question and answers are utilized to create the vocabulary of words to index mapping, which will be utilized for converting words into vector mappings:

```
# Creating Vocabulary
>>> import nltk
>>> import collections
>>> counter = collections.Counter()
>>> for i in range(len(QnAdata)):
...     for word in nltk.word_tokenize(QnAdata.iloc[i][2]):
...         counter[word]+=1
>>> word2idx = {w:(i+1) for i, (w,_) in enumerate(counter.most_common())}
>>> idx2word = {v:k for k,v in word2idx.items()}
>>> idx2word[0] = "PAD"
>>> vocab_size = len(word2idx)+1
>>> print (vocab_size)
```

Vocabulary size: 3451

Encoding and decoding functions are used to convert text to indices and indices to text respectively. As we know, Deep learning models work on numeric values rather than text or character data:

```
>>> def encode(sentence, maxlen, vocab_size):
...     indices = np.zeros((maxlen, vocab_size))
...     for i, w in enumerate(nltk.word_tokenize(sentence)):
...         if i == maxlen: break
...         indices[i, word2idx[w]] = 1
...     return indices
>>> def decode(indices, calc_argmax=True):
...     if calc_argmax:
...         indices = np.argmax(indices, axis=-1)
...     return ' '.join(idx2word[x] for x in indices)
```

The following code is used to vectorize the question and answers with the given maximum length for both questions and answers. Both might be different lengths. In some pieces of data, the question length is greater than answer length, and in a few cases, it's length is less than answer length. Ideally, the question length is good to catch the right answers. Unfortunately in this case, question length is much less than the answer length, which is a very bad example to develop generative models:

```
>>> question_maxlen = 10
>>> answer_maxlen = 20
>>> def create_questions(question_maxlen, vocab_size):
```

```

... question_idx = np.zeros(shape=(len(Questions), question_maxlen,
vocab_size))
... for q in range(len(Questions)):
...     question = encode(Questions[q], question_maxlen, vocab_size)
...     question_idx[i] = question
...     return question_idx
>>> quesns_train = create_questions(question_maxlen=question_maxlen,
vocab_size=vocab_size)
>>> def create_answers(answer_maxlen, vocab_size):
...     answer_idx = np.zeros(shape=(len(Answers), answer_maxlen, vocab_size))
...     for q in range(len(Answers)):
...         answer = encode(Answers[q], answer_maxlen, vocab_size)
...         answer_idx[i] = answer
...     return answer_idx
>>> answs_train = create_answers(answer_maxlen=answer_maxlen, vocab_size=
vocab_size)
>>> from keras.layers import Input, Dense, Dropout, Activation
>>> from keras.models import Model
>>> from keras.layers.recurrent import LSTM
>>> from keras.layers.wrappers import Bidirectional
>>> from keras.layers import RepeatVector, TimeDistributed,
ActivityRegularization

```

The following code is an important part of the chatbot. Here we have used recurrent networks, repeat vector, and time-distributed networks. The repeat vector used to match dimensions of input to output values. Whereas time-distributed networks are used to change the column vector to the output dimension's vocabulary size:

```

>>> n_hidden = 128
>>> question_layer = Input(shape=(question_maxlen, vocab_size))
>>> encoder_rnn = LSTM(n_hidden, dropout=0.2, recurrent_dropout=0.2)
(question_layer)
>>> repeat_encode = RepeatVector(answer_maxlen)(encoder_rnn)
>>> dense_layer = TimeDistributed(Dense(vocab_size))(repeat_encode)
>>> regularized_layer = ActivityRegularization(l2=1)(dense_layer)
>>> softmax_layer = Activation('softmax')(regularized_layer)
>>> model = Model([question_layer], [softmax_layer])
>>> model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
>>> print (model.summary())

```

The following model summary describes the change in flow of model size across the model. The input layer matches the question's dimension and the output matches the answer's dimension:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 10, 3451)	0
lstm_1 (LSTM)	(None, 128)	1832960
repeat_vector_1 (RepeatVecto	(None, 20, 128)	0
time_distributed_1 (TimeDist	(None, 20, 3451)	445179
activity_regularization_1 (A	(None, 20, 3451)	0
activation_1 (Activation)	(None, 20, 3451)	0
Total params: 2,278,139.0		
Trainable params: 2,278,139.0		
Non-trainable params: 0.0		

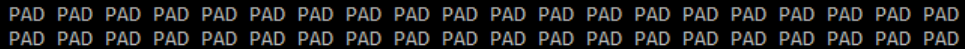
```
# Model Training
>>> quesns_train_2 = quesns_train.astype('float32')
>>> answs_train_2 = answs_train.astype('float32')
>>> model.fit(quesns_train_2, answs_train_2, batch_size=32, epochs=30,
validation_split=0.05)
```

The results are a bit tricky in the following screenshot even though the accuracy is significantly higher. The chatbot model might produce complete nonsense, as most of the words are padding here. The reason? The number of words in this data is less:

```
Train on 2803 samples, validate on 148 samples
Epoch 1/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 2/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 3/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 4/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 5/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 26/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 27/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 28/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 29/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Epoch 30/30
2803/2803 [=====] - 3s - loss: 0.0000e+00 - acc: 1.0000 - val_loss: 0.0571 - val_acc: 0.9932
Out[13]: <keras.callbacks.History at 0x17efbe358>
```

```
# Model prediction
>>> ans_pred = model.predict(quesns_train_2[0:3])
>>> print (decode(ans_pred[0]))
>>> print (decode(ans_pred[1]))
```

The following screenshot depicts the sample output on test data. The output does not seem to make sense, which is an issue with generative models:



```
PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD
PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD PAD
```

Our model did not work well in this case, but still some areas of improvement are possible going forward with generative chatbot models. Readers can give it a try:

- Have a dataset with lengthy questions and answers to catch signals well
- Create a larger architecture of deep learning models and train over longer iterations
- Make question-and-answer pairs more generic rather than factoid-based, such as retrieving knowledge and so on, where generative models fail miserably

Index

A

- abbreviations
 - making, in sentences 88, 89
- activation map size
 - calculating 233
- advanced sentiment analysis
 - exploring 218, 220, 221, 222, 223
- AlexNet 234
- anaphora
 - about 205
 - resolving 204, 205, 207, 208, 209
- applications, CNNs
 - face recognition 235
 - image classification 235
 - NLP 235
 - scene labeling 235
- Artificial Intelligence Markup Language (AIML) 283
- automated text generation
 - with long short-term memory (LSTM) 264, 265, 266

B

- Backus-Naur form (BNF) notation 108
- BeautifulSoup
 - used, for HTML parsing 50, 52, 53
- bi-grams 278
- bidirectional LSTM
 - IMDB sentiment classification 252, 254, 256
- Brown corpus
 - wh words, counting in genres 15, 17
- built-in chunker
 - using 122, 123, 124

C

- cataphora 205
- chart

- parsing 142, 144, 146
- chat text corpus files
 - frequency distribution operations, exploring on 17, 18, 20
- chunker
 - training 129, 130, 131, 132
 - writing 124, 125, 127, 128
- classification
 - used, for segmenting sentences 164, 166, 167
- contents
 - reading, from RSS feed 48, 50
- context
 - used, for writing POS tagger 173, 175, 176
- conversational assistant/chatbots
 - creating 223, 224, 226, 227, 228, 229
- Convolutional neural networks (CNNs) 1D
 - IMDB sentiment classification 247, 248, 249, 251
- Convolutional neural networks (CNNs)
 - about 231
 - advantages 231
 - applications 235
 - characteristics 231
- Cornell CS Movie review corpus 12

D

- date regex
 - creating 85, 86
- deep neural networks
 - used, for classifying emails 238, 239, 240, 243, 245
- dependency grammar
 - parsing 139, 141, 142
- dictionaries
 - creating 153, 154, 155
 - inversing 153, 155
 - using 156, 159

documents
 classifying 168, 170, 172

E

edit distance 66, 67, 69
emails
 classifying, deep neural networks used 238,
 239, 240, 243, 245
entities 148
external corpus
 accessing 14
 downloading 12
 loading 13

F

feature set
 selecting 159, 161, 163
features 159
five-character words
 finding 88, 89
frequency distribution operations
 exploring, on chat text corpus files 17, 18, 20
 exploring, on web corpus files 17, 18, 20

G

generative chatbot
 with recurrent neural networks (RNN) 283, 284,
 287
gensim library
 using 200
grammar
 writing 108, 109, 110, 111
group 87

H

high-dimensional words in 2D
 visualizing, with neural word vector visualization
 256, 258, 260, 262
HTML
 parsing, with BeautifulSoup 50, 52, 53
hypernym 25
hyponym 25

I

IMDB sentiment classification
 with bidirectional LSTM 252, 254, 256
 with convolutional networks (CNN) 1D 247, 248,
 249, 251
in-built corpora
 accessing 9, 10, 11
in-built tagger
 exploring 95, 96, 97
inbuilt NERs
 using 149, 150, 151, 152
inbuilt stemmers
 using, of NLTK 58, 59, 60
inbuilt tokenizers
 NLTK, using 55, 56, 57
inverse document frequency (IDF) 188

L

lemmatization 60, 62, 63
Levenshtein distance
 about 66
 reference 66
long short-term memory (LSTM)
 about 237
 using, in automated text generation 264, 265,
 266

M

many-to-many mappings 153
Mashable
 about 48
 reference link 48
memory networks
 questions and answers, on episodic data 270,
 271, 273, 276, 278
multiple literal strings
 searching 83, 84, 85

N

N-grams 278
named entities
 about 147, 148
 examples 148
Named Entity Recognition (NER) 149

- natural language processing (NLP) 8, 178
- neural word vector visualization
 - used, for visualizing high-dimensional words in 2D 256, 258, 259, 262
- NLP pipeline
 - creating 179, 180, 183, 184, 185, 187
- NLTK data
 - reference 9

O

- one-to-one mappings 153

P

- padding 233
- part-of-speech (POS) 94, 121, 147
- PDF file
 - reading, in Python 37, 38, 39
- polysemy
 - computing 28
- pooling 232
- POS tagger
 - writing, with context 173, 175, 176
- probabilistic CFG
 - writing 112, 113, 114, 116
- projective dependency
 - parsing 139, 140, 141, 142
- proper nouns 148
- PyPDF2 library
 - installing 37
- Python
 - about 153
 - PDF file, reading in 37, 38, 39
 - word documents, reading in 40, 41, 43, 44

R

- ranges of character
 - creating 85, 86
- recurrent neural networks (RNN)
 - about 235, 264
 - applications, in NLP 237
 - language modeling, for predicting next best word 278, 280, 282
 - used, for building generative chatbot 283, 284, 287
- recursive CFG

- writing 116, 117, 118, 119, 120
- recursive descent
 - parsing 132, 134, 135
- regex stemmer
 - writing 91, 93
- regex tokenizer
 - writing 89, 90, 91
- regular expressions
 - using 77, 78, 79, 81, 82
- Rich Site Summary (RSS) feed
 - about 48
 - contents, reading from 48, 50

S

- sentence
 - about 148
 - segmenting, classification used 164, 166, 167
- sentiment analysis
 - performing 214, 216, 217, 218
- set of characters
 - creating 85, 87
- shift-reduce
 - parsing 136, 138, 139
- stemming 58, 59, 60
- stopwords 63, 64, 66
- stories
 - processing, for common vocabulary extraction 69, 70, 73, 75
- string operations
 - exploring 34, 35, 37
 - importance 32, 33, 34
- substring occurrences
 - searching 83, 84, 85

T

- taggers
 - about 94
 - training 104, 106, 107
 - writing 98, 99, 101, 102, 103
- tagging 94
- term frequency (TF) 187
- text similarity problem
 - solving 187, 189, 190, 191, 192, 193
- text
 - summarizing 200, 202, 203, 204

TextRank algorithm

reference link 200

TF-IDF algorithm

used, for solving similarity problem 187

tokenization 55, 56, 57

topic identification 194, 196, 197, 198, 200

U

user-defined corpus

creating 44, 45, 46, 47

W

web corpus files

frequency distribution operations, exploring on

17, 19, 20

word documents

reading, in Python 40, 41, 43, 44

word sense

disambiguating 210, 211, 213

WordNet

hypernyms, exploring 25

hyponyms, exploring 25

polysemy, computing 29

used, for exploring senses of ambiguous word

20, 21, 24

WordnetLemmatizer

using, of NLTK 60, 62, 63