Robert Layton

Learning Data Mining with Python

Second Edition

Use Python to manipulate data and build predictive models

Packt

Learning Data Mining with Python

Second Edition

Use Python to manipulate data and build predictive models

Robert Layton

BIRMINGHAM - MUMBAI

Learning Data Mining with Python

Second Edition

First published: July 2015

Second edition: April 2017

Production reference: 1250417

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

ISBN 978-1-78712-678-7

www.packtpub.com

Credits

Author

Robert Layton

Reviewer

Asad Ahamad

Commissioning Editor

Veena Pagare

Acquisition Editor

Divya Poojari

Content Development Editor

Tejas Limkar

Technical Editor

Danish Shaikh

Copy Editor

Vikrant Phadkay

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Tania Dutta

Production Coordinator

Aparna Bhagat

About the Author

Robert Layton is a data scientist investigating data-driven applications to businesses across a number of sectors. He received a PhD investigating cybercrime analytics from the Internet Commerce Security Laboratory at Federation University Australia, before moving into industry, starting his own data analytics company dataPipeline (`www.datapipeline.com.au`). Next, he created Eureaktive (`www.eureaktive.com.au`), which works with tech-based startups on developing their proof-of-concepts and early-stage prototypes. Robert also runs `www.learningtensorflow.com`, which is one of the world's premier tutorial websites for Google's TensorFlow library.

Robert is an active member of the Python community, having used Python for more than 8 years. He has presented at PyConAU for the last four years and works with Python Charmers to provide Python-based training for businesses and professionals from a wide range of organisations.

Robert can be best reached via Twitter `@robertlayton`

Thank you to my family for supporting me on this journey, thanks to all the readers of revision 1 for making it a success, and thanks to Matty for his assistance behind-the-scenes with the book.

About the Reviewer

Asad Ahamad is a data enthusiast and loves to work on data to solve challenging problems.

He did his masters in Industrial Mathematics with Computer Application from Jamia Millia Islamia, New Delhi. He admires Mathematics a lot and always tries to use it to gain maximum profit for business.

He has good experience working on data mining, machine learning and data science and worked for various multinationals in India. He mainly uses R and Python to perform data wrangling and modeling. He is fond of using open source tools for data analysis.

He is active social media user. Feel free to connect him on twitter `@asadtaj88`

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/1787126781`.

If you'd like to join our team of regular reviewers, you can e-mail 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

The second revision of *Learning Data Mining with Python* was written with the programmer in mind. It aims to introduce data mining to a wide range of programmers, as I feel that this is critically important to all those in the computer science field. Data mining is quickly becoming the building block of the next generation of Artificial Intelligence systems. Even if you don't find yourself building these systems, you will be using them, interfacing with them, and being guided by them. Understand the process behind them is important and helps you get the best out of them.
The second revision builds upon the first. Many of chapters and exercises are similar, although new concepts are introduced and exercises are expanded in scope. Those that had read the first revision should be able to move quickly through the book and pick up new knowledge along the way and engage with the extra activities proposed. Those new to the book are encouraged to take their time, do the exercises and experiment. Feel free to break the code to understand it, and reach out if you have any questions.
As this is a book aimed at programmers, we assume that you have some knowledge of programming and of Python itself. For this reason, there is little explanation of what the *Python* code itself is doing, except in cases where it is ambiguous.

What this book covers

`Chapter 1`, *Getting started with data mining*, introduces the technologies we will be using, along with implementing two basic algorithms to get started.

`Chapter 2`, *Classifying with scikit-learn*, covers classification, a key form of data mining. You'll also learn about some structures for making your data mining experimentation easier to perform..

`Chapter 3`, *Predicting Sports Winners with Decisions Trees*, introduces two new algorithms, Decision Trees and Random Forests, and uses it to predict sports winners by creating useful features..

`Chapter 4`, *Recommending Movies using Affinity Analysis*, looks at the problem of recommending products based on past experience, and introduces the Apriori algorithm.

`Chapter 5`, *Features and scikit-learn Transformers*, introduces more types of features you can create, and how to work with different datasets.

`Chapter 6`, *Social Media Insight using Naive Bayes*, uses the Naïve Bayes algorithm to automatically parse text-based information from the social media website Twitter.

`Chapter 7`, *Follow Recommendations Using Graph Mining*, applies cluster analysis and network analysis to find good people to follow on social media.

`Chapter 8`, *Beating CAPTCHAs with Neural Networks*, looks at extracting information from images, and then training neural networks to find words and letters in those images.

`Chapter 9`, *Authorship attribution*, looks at determining who wrote a given documents, by extracting text-based features and using Support Vector Machines.

`Chapter 10`, *Clustering news articles*, uses the k-means clustering algorithm to group together news articles based on their content.

`Chapter 11`, *Object Detection in Images using Deep Neural Networks*, determines what type of object is being shown in an image, by applying deep neural networks.

`Chapter 12`, *Working with Big Data*, looks at workflows for applying algorithms to big data and how to get insight from it.

`Appendix`, *Next step*, goes through each chapter, giving hints on where to go next for a deeper understanding of the concepts introduced.

What you need for this book

It should come as no surprise that you'll need a computer, or access to one, to complete the book. The computer should be reasonably modern, but it doesn't need to be overpowered. Any modern processor (from about 2010 onwards) and 4 gigabytes of RAM will suffice, and you can probably run almost all of the code on a slower system too.

The exception here is with the final two chapters. In these chapters, I step through using Amazon's web services (AWS) for running the code. This will probably cost you some money, but the advantage is less system setup than running the code locally. If you don't want to pay for those services, the tools used can all be set-up on a local computer, but you will definitely need a modern system to run it. A processor built in at least 2012, and more than 4 GB of RAM are necessary.

I recommend the Ubuntu operating system, but the code should work well on Windows, Macs, or any other Linux variant. You may need to consult the documentation for your system to get some things installed though.

In this book, I use pip for installing code, which is a command line tool for installing Python libraries. Another option is to use Anaconda, which can be found online here: `http://continuum.io/downloads`

I also have tested all code using Python 3. Most of the code examples work on Python 2 with no changes. If you run into any problems, and can't get around it, send an email and we can offer a solution.

Who this book is for

This book is for programmers that want to get started in data mining in an application-focused manner.

If you haven't programmed before, I strongly recommend that you learn at least the basics before you get started. This book doesn't introduce programming, nor does it give too much time to explaining the actual implementation (in-code) of how to type out the instructions. That said, once you go through the basics, you should be able to come back to this book fairly quickly – there is no need to be an expert programmer first!

I highly recommend that you have some Python programming experience. If you don't, feel free to jump in, but you might want to take a look at some Python code first, possibly focused on tutorials using the IPython notebook. Writing programs in the IPython notebook works a little differently than other methods, such as writing a Java program in a fully-fledged IDE.

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: "The next lines of code read the link and assign it to the to the `dataset_filename` function."

A block of code is set as follows:

```
import numpy as np
dataset_filename = "affinity_dataset.txt"
X = np.loadtxt(dataset_filename)
```

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

```
$ conda install scikit-learn
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**."

Warnings or important notes appear in a box 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**.

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/Learning-Data-Mining-with-Python-Second-Edition`. The benefit of the github repository is that any issues with the code, including problems relating to software version changes, will be kept track of and the code there will include changes from readers around the world. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

To avoid indention issues please use the code bundle to run the codes in the IDE instead of copying directly from the PDF.

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
Getting Started with Data Mining

We are collecting information about our world on a scale that has never been seen before in the history of humanity. Along with this trend, we are now placing more day-to-day importance on the use of this information in everyday life. We now expect our computers to translate web pages into other languages, predict the weather with high accuracy, suggest books we would like, and to diagnose our health issues. These expectations will grow into the future, both in application breadth and efficacy. **Data Mining** is a methodology that we can employ to train computers to make decisions with data and forms the backbone of many high-tech systems of today.

The **Python** programming language is fast growing in popularity, for a good reason. It gives the programmer flexibility, it has many modules to perform different tasks, and Python code is usually more readable and concise than in any other languages. There is a large and an active community of researchers, practitioners, and beginners using Python for data mining.

In this chapter, we will introduce data mining with Python. We will cover the following topics

- What is data mining and where can we use it?
- Setting up a Python-based environment to perform data mining
- An example of affinity analysis, recommending products based on purchasing habits
- An example of (a classic) classification problem, predicting the plant species based on its measurement

Introducing data mining

Data mining provides a way for a computer to learn how to make decisions with data. This decision could be predicting tomorrow's weather, blocking a spam email from entering your inbox, detecting the language of a website, or finding a new romance on a dating site. There are many different applications of data mining, with new applications being discovered all the time.

Data mining is part algorithm design, statistics, engineering, optimization, and computer science. However, combined with these *base* skills in the area, we also need to apply **domain knowledge (expert knowledge)**of the area we are applying the data mining. Domain knowledge is critical for going from good results to great results. Applying data mining effectively usually requires this domain-specific knowledge to be integrated with the algorithms.

Most data mining applications work with the same **high-level** view, where a model learns from some data and is applied to other data, although the details often change quite considerably.

Data mining applications involve creating data sets and tuning the algorithm as explained in the following steps

1. We start our data mining process by creating a dataset, describing an aspect of the real world. Datasets comprise of the following two aspects:

    - **Samples**: These are objects in the real world, such as a book, photograph, animal, person, or any other object. Samples are also referred to as observations, records or rows, among other naming conventions.
    - **Features**: These are descriptions or measurements of the samples in our dataset. Features could be the length, frequency of a specific word, the number of legs on an animal, date it was created, and so on. Features are also referred to as variables, columns, attributes or covariant, again among other naming conventions.

2. The next step is tuning the data mining algorithm. Each data mining algorithm has parameters, either within the algorithm or supplied by the user. This tuning allows the algorithm to learn how to make decisions about the data.

As a simple example, we may wish the computer to be able to categorize people as *short* or *tall*. We start by collecting our dataset, which includes the heights of different people and whether they are considered short or tall:

Person	Height	Short or tall?
1	155cm	Short
2	165cm	Short
3	175cm	Tall
4	185cm	Tall

As explained above, the next step involves tuning the parameters of our algorithm. As a simple algorithm; if the height is more than x, the person is tall. Otherwise, they are short. Our training algorithms will then look at the data and decide on a good value for x. For the preceding data, a reasonable value for this threshold would be 170 cm. A person taller than 170 cm is considered tall by the algorithm. Anyone else is considered short by this measure. This then lets our algorithm classify new data, such as a person with height 167 cm, even though we may have never seen a person with those measurements before.

In the preceding data, we had an obvious feature type. We wanted to know if people are short or tall, so we collected their heights. This feature engineering is a critical problem in data mining. In later chapters, we will discuss methods for choosing good features to collect in your dataset. Ultimately, this step often requires some expert domain knowledge or at least some trial and error.

In this book, we will introduce data mining through Python. In some cases, we choose clarity of code and workflows, rather than the most optimized way to perform every task. This clarity sometimes involves skipping some details that can improve the algorithm's speed or effectiveness.

Using Python and the Jupyter Notebook

In this section, we will cover installing Python and the environment that we will use for most of the book, the **Jupyter** Notebook. Furthermore, we will install the **NumPy** module, which we will use for the first set of examples.

The Jupyter Notebook was, until very recently, called the IPython Notebook. You'll notice the term in web searches for the project. Jupyter is the new name, representing a broadening of the project beyond using just Python.

Installing Python

The Python programming language is a fantastic, versatile, and an easy to use language.

For this book, we will be using Python 3.5, which is available for your system from the Python Organization's website `https://www.python.org/downloads/`. However, I recommend that you use Anaconda to install Python, which you can download from the official website at `https://www.continuum.io/downloads`.

There will be two major versions to choose from, Python 3.5 and Python 2.7. Remember to download and install Python 3.5, which is the version tested throughout this book. Follow the installation instructions on that website for your system. If you have a strong reason to learn version 2 of Python, then do so by downloading the Python 2.7 version. Keep in mind that some code may not work as in the book, and some workarounds may be needed.

In this book, I assume that you have some knowledge of programming and Python itself. You do not need to be an expert with Python to complete this book, although a good level of knowledge will help. I will not be explaining general code structures and syntax in this book, except where it is different from what is considered *normal* python coding practice.

If you do not have any experience with programming, I recommend that you pick up the *Learning Python* book from Packt Publishing, or the book *Dive Into Python*, available online at `www.diveintopython3.net`

The Python organization also maintains a list of two online tutorials for those new to Python:

- For non-programmers who want to learn to program through the Python language:

```
https://wiki.python.org/moin/BeginnersGuide/NonProgrammers
```

- For programmers who already know how to program, but need to learn Python specifically:

```
https://wiki.python.org/moin/BeginnersGuide/Programmers
```

Windows users will need to set an environment variable to use Python from the command line, where other systems will usually be immediately executable. We set it in the following steps

1. First, find where you install Python 3 onto your computer; the default location is `C:\Python35`.
2. Next, enter this command into the command line (cmd program): set the environment to `PYTHONPATH=%PYTHONPATH%;C:\Python35`.

Remember to change the `C:\Python35` if your installation of Python is in a different folder.

Once you have Python running on your system, you should be able to open a command prompt and can run the following code to be sure it has installed correctly.

```
    $ python
    Python 3.5.1 (default, Apr 11 2014, 13:05:11)
    [GCC 4.8.2] on Linux
    Type "help", "copyright", "credits" or "license" for more
      information.
    >>> print("Hello, world!")
Hello, world!
    >>> exit()
```

Note that we will be using the dollar sign ($) to denote that a command that you type into the terminal (also called a shell or `cmd` on Windows). You do not need to type this character (or retype anything that already appears on your screen). Just type in the rest of the line and press Enter.

After you have the above `"Hello, world!"` example running, exit the program and move on to installing a more advanced environment to run Python code, the Jupyter Notebook.

Python 3.5 will include a program called **pip**, which is a package manager that helps to install new libraries on your system. You can verify that `pip` is working on your system by running the `$ pip freeze` command, which tells you which packages you have installed on your system. Anaconda also installs their package manager, `conda`, that you can use. If unsure, use `conda` first, use `pip` only if that fails.

Installing Jupyter Notebook

Jupyter is a platform for Python development that contains some tools and environments for running Python and has more features than the standard interpreter. It contains the powerful Jupyter Notebook, which allows you to write programs in a web browser. It also formats your code, shows output, and allows you to annotate your scripts. It is a great tool for exploring datasets and we will be using it as our main environment for the code in this book.

To install the Jupyter Notebook on your computer, you can type the following into a command line prompt (not into Python):

```
$ conda install jupyter notebook
```

You will not need administrator privileges to install this, as Anaconda keeps packages in the user's directory.

With the Jupyter Notebook installed, you can launch it with the following:

```
$ jupyter notebook
```

Running this command will do two things. First, it will create a Jupyter Notebook instance - the backend - that will run in the command prompt you just used. Second, it will launch your web browser and connect to this instance, allowing you to create a new notebook. It will look something like the following screenshot (where you need to replace `/home/bob` with your current working directory):

To stop the Jupyter Notebook from running, open the command prompt that has the instance running (the one you used earlier to run the `jupyter notebook` command). Then, press *Ctrl* + *C* and you will be prompted `Shutdown this notebook server (y/[n])?`. Type *y* and press *Enter* and the Jupyter Notebook will shut down.

Installing scikit-learn

The `scikit-learn` package is a machine learning library, written in Python (but also containing code in other languages). It contains numerous algorithms, datasets, utilities, and frameworks for performing machine learning. Scikit-learnis built upon the scientific python stack, including libraries such as the `NumPy` and `SciPy` for speed. Scikit-learn is fast and scalable in many instances and useful for all skill ranges from beginners to advanced research users. We will cover more details of scikit-learn in `Chapter 2`, *Classifying with scikit-learn Estimators.*

To install `scikit-learn`, you can use the `conda` utility that comes with Python 3, which will also install the `NumPy` and `SciPy` libraries if you do not already have them. Open a terminal with administrator/root privileges and enter the following command:

```
$ conda install scikit-learn
```

Users of major Linux distributions such as Ubuntu or Red Hat may wish to install the official package from their package manager.

Not all distributions have the latest versions of scikit-learn, so check the version before installing it. The minimum version needed for this book is 0.14. My recommendation for this book is to use Anaconda to manage this for you, rather than installing using your system's package manager.

Those wishing to install the latest version by compiling the source, or view more detailed installation instructions, can go to `http://scikit-learn.org/stable/install.html` and refer the official documentation on installing scikit-learn.

A simple affinity analysis example

In this section, we jump into our first example. A common use case for data mining is to improve sales, by asking a customer who is buying a product if he/she would like another similar product as well. You can perform this analysis through affinity analysis, which is the study of when things exist together, namely. correlate to each other.

To repeat the now-infamous phrase taught in statistics classes, *correlation is not causation*. This phrase means that the results from affinity analysis cannot give a cause. In our next example, we perform affinity analysis on product purchases. The results indicate that the products are purchased together, but not that buying one product causes the purchase of the other. The distinction is important, critically so when determining how to use the results to affect a business process, for instance.

What is affinity analysis?

Affinity analysis is a type of data mining that gives similarity between samples (objects). This could be the similarity between the following:

- **Users** on a website, to provide varied services or targeted advertising
- **Items** to sell to those users, to provide recommended movies or products
- **Human genes**, to find people that share the same ancestors

We can measure affinity in several ways. For instance, we can record how frequently two products are purchased together. We can also record the accuracy of the statement when a person buys object 1 and when they buy object 2. Other ways to measure affinity include computing the similarity between samples, which we will cover in later chapters.

Product recommendations

One of the issues with moving a traditional business online, such as commerce, is that tasks that used to be done by humans need to be automated for the online business to scale and compete with existing automated businesses. One example of this is up-selling, or selling an extra item to a customer who is already buying. Automated product recommendations through data mining are one of the driving forces behind the e-commerce revolution that is turning billions of dollars per year into revenue.

In this example, we are going to focus on a basic product recommendation service. We design this based on the following idea: when two items are historically purchased together, they are more likely to be purchased together in the future. This sort of thinking is behind many product recommendation services, in both online and offline businesses.

A very simple algorithm for this type of product recommendation algorithm is to simply find any historical case where a user has brought an item and to recommend other items that the historical user brought. In practice, simple algorithms such as this can do well, at least better than choosing random items to recommend. However, they can be improved upon significantly, which is where data mining comes in.

To simplify the coding, we will consider only two items at a time. As an example, people may buy bread and milk at the same time at the supermarket. In this early example, we wish to find simple rules of the form:

If a person buys product X, then they are likely to purchase product Y

More complex rules involving multiple items will not be covered such as people buying sausages and burgers being more likely to buy tomato sauce.

Loading the dataset with NumPy

The dataset can be downloaded from the code package supplied with the book, or from the official GitHub repository at:

`https://github.com/dataPipelineAU/LearningDataMiningWithPython2`

Download this file and save it on your computer, noting the path to the dataset. It is easiest to put it in the directory you'll run your code from, but we can load the dataset from anywhere on your computer.

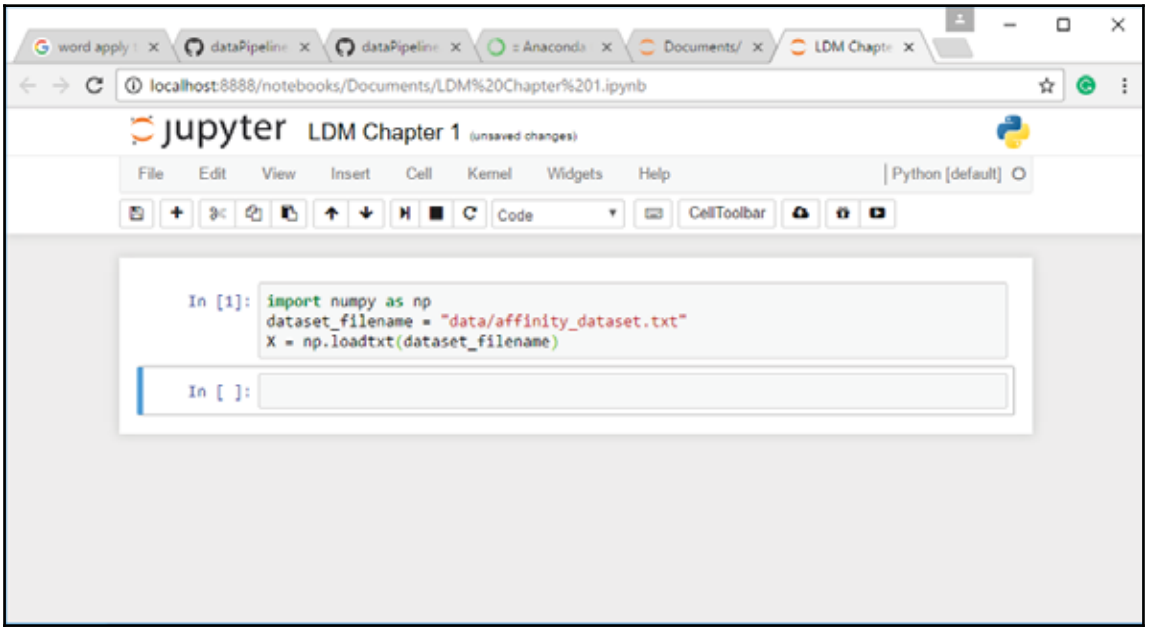
For this example, I recommend that you create a new folder on your computer to store your dataset and code. From here, open your Jupyter Notebook, navigate to this folder, and create a new notebook.

The dataset we are going to use for this example is a NumPy two-dimensional array, which is a format that underlies most of the examples in the rest of the book. The array looks like a table, with rows representing different samples and columns representing different features.

The cells represent the value of a specific feature of a specific sample. To illustrate, we can load the dataset with the following code:

```
import numpy as np
dataset_filename = "affinity_dataset.txt"
X = np.loadtxt(dataset_filename)
```

Enter the previous code into the first cell of your (Jupyter) Notebook. You can then run the code by pressing Shift + Enter (which will also add a new cell for the next section of code). After the code is run, the square brackets to the left-hand side of the first cell will be assigned an incrementing number, letting you know that this cell has completed. The first cell should look like the following:



For code that will take more time to run, an asterisk will be placed here to denote that this code is either running or scheduled to run. This asterisk will be replaced by a number when the code has completed running (including if the code completes because it failed).

This dataset has 100 samples and five features, which we will need to know for the later code. Let's extract those values using the following code:

```
n_samples, n_features = X.shape
```

If you choose to store the dataset somewhere other than the directory your Jupyter Notebooks are in, you will need to change the `dataset_filename` value to the new location.

Next, we can show some of the rows of the dataset to get an understanding of the data. Enter the following line of code into the next cell and run it, to print the first five lines of the dataset:

```
print(X[:5])
```

The result will show you which items were bought in the first five transactions listed:

```
[[ 0.  1.  0.  0.  0.]
 [ 1.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  1.]
 [ 1.  1.  0.  0.  0.]
 [ 0.  0.  1.  1.  1.]]
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you could visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you. I've also setup a GitHub repository that contains a live version of the code, along with new fixes, updates and so on. You can retrieve the code and datasets at the repository here:
`https://github.com/dataPipelineAU/LearningDataMiningWithPython2`

You can read the dataset can by looking at each row (horizontal line) at a time. The first row `(0, 1, 0, 0, 0)` shows the items purchased in the first transaction. Each column (vertical row) represents each of the items. They are bread, milk, cheese, apples, and bananas, respectively. Therefore, in the first transaction, the person bought cheese, apples, and bananas, but not bread or milk. Add the following line in a new cell to allow us to turn these feature numbers into actual words:

```
features = ["bread", "milk", "cheese", "apples", "bananas"]
```

Each of these features contains binary values, stating only whether the items were purchased and not how many of them were purchased. A *1* indicates that *at least 1* item was bought of this type, while a *0* indicates that absolutely none of that item was purchased. For a real world dataset, using exact figures or a larger threshold would be required.

Implementing a simple ranking of rules

We wish to find rules of the type *If a person buys product X, then they are likely to purchase product Y*. We can quite easily create a list of all the rules in our dataset by simply finding all occasions when two products are purchased together. However, we then need a way to determine good rules from bad ones allowing us to choose specific products to recommend.

We can evaluate rules of this type in many ways, on which we will focus on two: **support** and **confidence.**

Support is the number of times that a rule occurs in a dataset, which is computed by simply counting the number of samples for which the rule is valid. It can sometimes be normalized by dividing by the total number of times the premise of the rule is valid, but we will simply count the total for this implementation.

The **premise** is the requirements for a rule to be considered active. The **conclusion** is the output of the rule. For the example *if a person buys an apple, they also buy a banana*, the rule is only valid if the premise happens - a person buys an apple. The rule's conclusion then states that the person will buy a banana.

While the support measures how often a rule exists, confidence measures how accurate they are when they can be used. You can compute this by determining the percentage of times the rule applies when the premise applies. We first count how many times a rule applies to our data and divide it by the number of samples where the premise (the `if` statement) occurs.

As an example, we will compute the support and confidence for the rule *if a person buys apples, they also buy bananas.*

As the following example shows, we can tell whether someone bought apples in a transaction, by checking the value of `sample[3]`, where we assign a sample to a row of our matrix:

```
sample = X[2]
```

Similarly, we can check if bananas were bought in a transaction by seeing if the value of `sample[4]` is equal to 1 (and so on). We can now compute the number of times our rule exists in our dataset and, from that, the confidence and support.

Now we need to compute these statistics for all rules in our database. We will do this by creating a dictionary for both *valid rules* and *invalid rules*. The key to this dictionary will be a tuple (premise and conclusion). We will store the indices, rather than the actual feature names. Therefore, we would store (3 and 4) to signify the previous rule *If a person buys apples, they will also buy bananas.* If the premise and conclusion are given, the rule is considered valid. While if the premise is given but the conclusion is not, the rule is considered invalid for that sample.

The following steps will help us to compute the confidence and support for all possible rules:

1. We first set up some dictionaries to store the results. We will use `defaultdict` for this, which sets a default value if a key is accessed that doesn't yet exist. We record the number of valid rules, invalid rules, and occurrences of each premise:

```
from collections import defaultdict
valid_rules = defaultdict(int)
invalid_rules = defaultdict(int)
num_occurences = defaultdict(int)
```

2. Next, we compute these values in a large loop. We iterate over each sample in the dataset and then loop over each feature as a premise. When again loop over each feature as a possible conclusion, mapping the relationship premise to conclusion. If the sample contains a person who bought the premise and the conclusion, we record this in `valid_rules`. If they did not purchase the conclusion product, we record this in `invalid_rules`.
3. For sample in X:

```
for sample in X:
    for premise in range(n_features):
    if sample[premise] == 0: continue
# Record that the premise was bought in another transaction
    num_occurences[premise] += 1
    for conclusion in range(n_features):
    if premise == conclusion:
# It makes little sense to
    measure if X -> X.
    continue
    if sample[conclusion] == 1:
# This person also bought the conclusion item
```

```
                valid_rules[(premise, conclusion)] += 1
```

If the premise is valid for this sample (it has a value of *1*), then we record this and check each conclusion of our rule. We skip over any conclusion that is the same as the premise-this would give us rules such as: *if a person buys Apples, then they buy Apples*, which obviously doesn't help us much.

We have now completed computing the necessary statistics and can now compute the *support* and *confidence* for each rule. As before, the support is simply our `valid_rules` value:

```
support = valid_rules
```

We can compute the confidence in the same way, but we must loop over each rule to compute this:

```
confidence = defaultdict(float)
for premise, conclusion in valid_rules.keys():
    rule = (premise, conclusion)
    confidence[rule] = valid_rules[rule] / num_occurences [premise]
```

We now have a dictionary with the support and confidence for each rule. We can create a function that will print out the rules in a readable format. The signature of the rule takes the premise and conclusion indices, the support and confidence dictionaries we just computed, and the features array that tells us what the `features` mean. Then we print out the `Support` and `Confidence` of this rule:

```
for premise, conclusion in confidence:
    premise_name = features[premise]
    conclusion_name = features[conclusion]
    print("Rule: If a person buys {0} they will also
          buy{1}".format(premise_name, conclusion_name))
    print(" - Confidence: {0:.3f}".format
          (confidence[(premise,conclusion)]))
    print(" - Support: {0}".format(support
                                   [(premise,
                                     conclusion)]))
    print("")
```

We can test the code by calling it in the following way-feel free to experiment with different premises and conclusions:

```
for premise, conclusion in confidence:
    premise_name = features[premise]
    conclusion_name = features[conclusion]
    print("Rule: If a person buys {0} they will also
          buy{1}".format(premise_name, conclusion_name))
```

```
        print(" - Confidence: {0:.3f}".format
              (confidence[(premise,conclusion)]))
        print(" - Support: {0}".format(support
                                       [(premise,
                                         conclusion)]))
        print("")
```

Ranking to find the best rules

Now that we can compute the support and confidence of all rules, we want to be able to find the *best* rules. To do this, we perform a ranking and print the ones with the highest values. We can do this for both the support and confidence values.

To find the rules with the highest support, we first sort the support dictionary. Dictionaries do not support ordering by default; the `items()` function gives us a list containing the data in the dictionary. We can sort this list using the `itemgetter` class as our key, which allows for the sorting of nested lists such as this one. Using `itemgetter(1)` allows us to sort based on the values. `Setting reverse=True` gives us the highest values first:

```
from operator import itemgetter
sorted_support = sorted(support.items(), key=itemgetter(1), reverse=True)
```

We can then print out the top five rules:

```
sorted_confidence = sorted(confidence.items(), key=itemgetter(1),
                           reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

The result will look like the following:

```
Rule #1
Rule: If a person buys bananas they will also buy milk
 - Support: 27
 - Confidence: 0.474
Rule #2
Rule: If a person buys milk they will also buy bananas
 - Support: 27
 - Confidence: 0.519
Rule #3
Rule: If a person buys bananas they will also buy apples
 - Support: 27
 - Confidence: 0.474
Rule #4
```

```
Rule: If a person buys apples they will also buy bananas
 - Support: 27
 - Confidence: 0.628
Rule #5
Rule: If a person buys apples they will also buy cheese
 - Support: 22
 - Confidence: 0.512
```

Similarly, we can print the top rules based on confidence. First, compute the sorted confidence list and then print them out using the same method as before.

```
sorted_confidence = sorted(confidence.items(), key=itemgetter(1),
                           reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

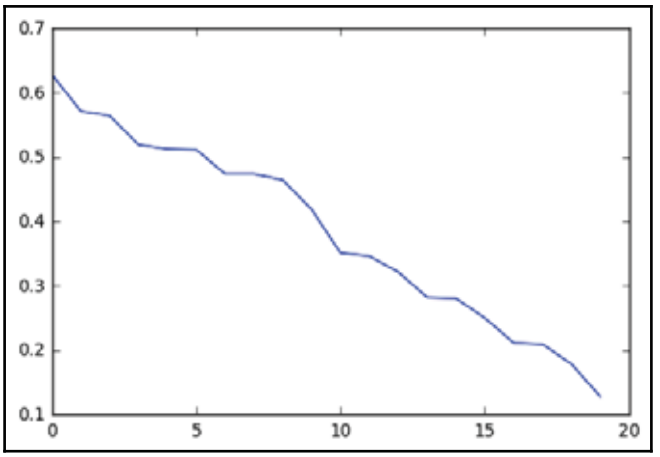
Two rules are near the top of both lists. The first is *If a person buys apples, they will also buy cheese*, and the second is *If a person buys cheese, they will also buy bananas*. A store manager can use rules like these to organize their store. For example, if apples are on sale this week, put a display of cheeses nearby. Similarly, it would make little sense to put both bananas on sale at the same time as cheese, as nearly 66 percent of people buying cheese will probably buy bananas -our sale won't increase banana purchases all that much.

Jupyter Notebook will display graphs inline, right in the notebook. Sometimes, however, this is not always configured by default. To configure Jupyter Notebook to display graphs inline, use the following line of code: `%matplotlib inline`

We can visualize the results using a library called `matplotlib`.

We are going to start with a simple line plot showing the confidence values of the rules, in order of confidence. `matplotlib` makes this easy - we just pass in the numbers, and it will draw up a simple but effective plot:

```
from matplotlib import pyplot as plt
plt.plot([confidence[rule[0]] for rule in sorted_confidence])
```



Using the previous graph, we can see that the first five rules have decent confidence, but the efficacy drops quite quickly after that. Using this information, we might decide to use just the first five rules to drive business decisions. Ultimately with exploration techniques like this, the result is up to the user.

Data mining has great exploratory power in examples like this. A person can use data mining techniques to explore relationships within their datasets to find new insights. In the next section, we will use data mining for a different purpose: prediction and classification.

A simple classification example

In the affinity analysis example, we looked for correlations between different variables in our dataset. In classification, we have a single variable that we are interested in and that we call the **class** (also called the target). In the earlier example, if we were interested in how to make people buy more apples, we would explore the rules related to apples and use those to inform our decisions.

What is classification?

Classification is one of the largest uses of data mining, both in practical use and in research. As before, we have a set of samples that represents objects or things we are interested in classifying. We also have a new array, the class values. These class values give us a categorization of the samples. Some examples are as follows:

- Determining the species of a plant by looking at its measurements. The class value here would be: *Which species is this?*
- Determining if an image contains a dog. The class would be: *Is there a dog in this image?*
- Determining if a patient has cancer, based on the results of a specific test. The class would be: *Does this patient have cancer?*

While many of the examples previous are binary (yes/no) questions, they do not have to be, as in the case of plant species classification in this section.

The goal of classification applications is to train a model on a set of samples with known classes and then apply that model to new unseen samples with unknown classes. For example, we want to train a spam classifier on my past e-mails, which I have labeled as spam or not spam. I then want to use that classifier to determine whether my next email is spam, without me needing to classify it myself.

Loading and preparing the dataset

The dataset we are going to use for this example is the famous Iris database of plant classification. In this dataset, we have 150 plant samples and four measurements of each: **sepal length**, **sepal width**, **petal length**, and **petal width** (all in centimeters). This classic dataset (first used in 1936!) is one of the classic datasets for data mining. There are three classes: **Iris Setosa**, **Iris Versicolour**, and **Iris Virginica**. The aim is to determine which type of plant a sample is, by examining its measurements.

The `scikit-learn` library contains this dataset built-in, making the loading of the dataset straightforward:

```
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

You can also `print(dataset.DESCR)` to see an outline of the dataset, including some details about the features.

The features in this dataset are continuous values, meaning they can take any range of values. Measurements are a good example of this type of feature, where a measurement can take the value of 1, 1.2, or 1.25 and so on. Another aspect of continuous features is that feature values that are close to each other indicate similarity. A plant with a sepal length of 1.2 cm is like a plant with a Sepal width of 1.25 cm.

In contrast are categorical features. These features, while often represented as numbers, cannot be compared in the same way. In the Iris dataset, the class values are an example of a categorical feature. The class 0 represents Iris Setosa; class 1 represents Iris Versicolour, and class 2 represents Iris Virginica. The numbering doesn't mean that Iris Setosa is more similar to Iris Versicolour than it is to Iris Virginica-despite the class value being more similar. The numbers here represent categories. All we can say is whether categories are the same or different.

There are other types of features too, which we will cover in later chapters. These include pixel intensity, word frequency and n-gram analysis.

While the features in this dataset are continuous, the algorithm we will use in this example requires categorical features. Turning a continuous feature into a categorical feature is a process called discretization.

A simple discretization algorithm is to choose some threshold, and any values below this threshold are given a value 0. Meanwhile, any above this are given the value 1. For our threshold, we will compute the mean (average) value for that feature. To start with, we compute the mean for each feature:

```
attribute_means = X.mean(axis=0)
```

The result from this code will be an array of length 4, which is the number of features we have. The first value is the mean of the values for the first feature and so on. Next, we use this to transform our dataset from one with continuous features to one with discrete categorical features:

```
assert attribute_means.shape == (n_features,)
X_d = np.array(X >= attribute_means, dtype='int')
```

We will use this new `X_d` dataset (for *X discretized*) for our **training and testing**, rather than the original dataset (*X*).

Implementing the OneR algorithm

OneR is a simple algorithm that simply predicts the class of a sample by finding the most frequent class for the feature values. **OneR** is shorthand for *One Rule*, indicating we only use a single rule for this classification by choosing the feature with the best performance. While some of the later algorithms are significantly more complex, this simple algorithm has been shown to have good performance in some real-world datasets.

The algorithm starts by iterating over every value of every feature. For that value, count the number of samples from each class that has that feature value. Record the most frequent class of the feature value, and the error of that prediction.

For example, if a feature has two values, *0* and *1*, we first check all samples that have the value *0*. For that value, we may have 20 in Class *A*, 60 in Class *B*, and a further 20 in Class *C*. The most frequent class for this value is *B*, and there are 40 instances that have different classes. The prediction for this feature value is *B* with an error of 40, as there are 40 samples that have a different class from the prediction. We then do the same procedure for the value *1* for this feature, and then for all other feature value combinations.

Once these combinations are computed, we compute the error for each feature by summing up the errors for all values for that feature. The feature with the lowest total error is chosen as the *One Rule* and then used to classify other instances.

In code, we will first create a function that computes the class prediction and error for a specific feature value. We have two necessary imports, `defaultdict` and `itemgetter`, that we used in earlier code:

```
from collections import defaultdict
from operator import itemgetter
```

Next, we create the function definition, which needs the dataset, classes, the index of the feature we are interested in, and the value we are computing. It loops over each sample, and counts the number of time each feature value corresponds to a specific class. We then choose the most frequent class for the current feature/value pair:

```
def train_feature_value(X, y_true, feature, value):
# Create a simple dictionary to count how frequency they give certain
predictions
 class_counts = defaultdict(int)
# Iterate through each sample and count the frequency of each
class/value pair
 for sample, y in zip(X, y_true):
    if sample[feature] == value:
        class_counts[y] += 1
# Now get the best one by sorting (highest first) and choosing the
```

```
first item
sorted_class_counts = sorted(class_counts.items(), key=itemgetter(1),
                             reverse=True)
most_frequent_class = sorted_class_counts[0][0]
 # The error is the number of samples that do not classify as the most
frequent class
 # *and* have the feature value.
    n_samples = X.shape[1]
    error = sum([class_count for class_value, class_count in
                 class_counts.items()
 if class_value != most_frequent_class])
    return most_frequent_class, error
```

As a final step, we also compute the error of this rule. In the `OneR` algorithm, any sample with this feature value would be predicted as being the most frequent class. Therefore, we compute the error by summing up the counts for the other classes (not the most frequent). These represent training samples that result in error or an incorrect classification.

With this function, we can now compute the error for an entire feature by looping over all the values for that feature, summing the errors, and recording the predicted classes for each value.

The function needs the dataset, classes, and feature index we are interested in. It then iterates through the different values and finds the most accurate feature value to use for this specific feature, as the rule in `OneR`:

```
def train(X, y_true, feature):
    # Check that variable is a valid number
    n_samples, n_features = X.shape
    assert 0 <= feature < n_features
    # Get all of the unique values that this variable has
    values = set(X[:,feature])
    # Stores the predictors array that is returned
    predictors = dict()
    errors = []
    for current_value in values:
        most_frequent_class, error = train_feature_value
        (X, y_true, feature, current_value)
        predictors[current_value] = most_frequent_class
        errors.append(error)
    # Compute the total error of using this feature to classify on
    total_error = sum(errors)
    return predictors, total_error
```

Let's have a look at this function in a little more detail.

After some initial tests, we then find all the unique values that the given feature takes. The indexing in the next line looks at the whole column for the given feature and returns it as an array. We then use the set function to find only the unique values:

```
    values = set(X[:,feature_index])
```

Next, we create our dictionary that will store the predictors. This dictionary will have feature values as the keys and classification as the value. An entry with key 1.5 and value 2 would mean that, when the feature has a value set to 1.5, classify it as belonging to class 2. We also create a list storing the errors for each feature value:

```
predictors = {}
    errors = []
```

As the main section of this function, we iterate over all the unique values for this feature and use our previously defined `train_feature_value` function to find the most frequent class and the error for a given feature value. We store the results as outlined earlier:

Finally, we compute the total errors of this rule and return the predictors along with this value:

```
total_error = sum(errors)
return predictors, total_error
```

Testing the algorithm

When we evaluated the affinity analysis algorithm of the earlier section, our aim was to explore the current dataset. With this classification, our problem is different. We want to build a model that will allow us to classify previously unseen samples by comparing them to what we know about the problem.

For this reason, we split our machine-learning workflow into two stages: training and testing. In training, we take a portion of the dataset and create our model. In testing, we apply that model and evaluate how effectively it worked on the dataset. As our goal is to create a model that can classify previously unseen samples, we cannot use our testing data for training the model. If we do, we run the risk of **overfitting**.

Overfitting is the problem of creating a model that classifies our training dataset very well but performs poorly on new samples. The solution is quite simple: never use training data to test your algorithm. This simple rule has some complex variants, which we will cover in later chapters; but, for now, we can evaluate our `OneR` implementation by simply splitting our dataset into two small datasets: a training one and a testing one. This workflow is given in this section.

The `scikit-learn` library contains a function to split data into training and testing components:

```
from sklearn.cross_validation import train_test_split
```

This function will split the dataset into two sub-datasets, per a given ratio (which by default uses 25 percent of the dataset for testing). It does this randomly, which improves the confidence that the algorithm will perform as expected in real world environments (where we expect data to come in from a random distribution):

```
Xd_train, Xd_test, y_train, y_test = train_test_split(X_d, y,
    random_state=14)
```

We now have two smaller datasets: `Xd_train` contains our data for training and `Xd_test` contains our data for testing. `y_train` and `y_test` give the corresponding class values for these datasets.

We also specify a `random_state`. Setting the random state will give the same split every time the same value is entered. It will *look* random, but the algorithm used is deterministic, and the output will be consistent. For this book, I recommend setting the random state to the same value that I do, as it will give you the same results that I get, allowing you to verify your results. To get truly random results that change every time you run it, set `random_state` to `None`.

Next, we compute the predictors for all the features for our dataset. Remember to only use the training data for this process. We iterate over all the features in the dataset and use our previously defined functions to train the predictors and compute the errors:

```
all_predictors = {}
errors = {}
for feature_index in range(Xd_train.shape[1]):
    predictors, total_error = train(Xd_train,
                                    y_train,
                                    feature_index)
    all_predictors[feature_index] = predictors
    errors[feature_index] = total_error
```

Next, we find the best feature to use as our *One Rule*, by finding the feature with the lowest error:

```
best_feature, best_error = sorted(errors.items(), key=itemgetter(1))[0]
```

We then create our `model` by storing the predictors for the best feature:

```
model = {'feature': best_feature,
         'predictor': all_predictors[best_feature]}
```

Our model is a dictionary that tells us which feature to use for our *One Rule* and the predictions that are made based on the values it has. Given this model, we can predict the class of a previously unseen sample by finding the value of the specific feature and using the appropriate predictor. The following code does this for a given sample:

```
variable = model['feature']
predictor = model['predictor']
prediction = predictor[int(sample[variable])]
```

Often we want to predict several new samples at one time, which we can do using the following function. It simply uses the above code, but iterate over all the samples in a dataset, obtaining the prediction for each sample:

```
def predict(X_test, model):
variable = model['feature']
predictor = model['predictor']
y_predicted = np.array([predictor
                        [int(sample[variable])] for sample
                        in X_test])
return y_predicted
```

For our `testing` dataset, we get the predictions by calling the following function:

```
y_predicted = predict(Xd_test, model)
```

We can then compute the accuracy of this by comparing it to the known classes:

```
accuracy = np.mean(y_predicted == y_test) * 100
print("The test accuracy is {:.1f}%".format(accuracy))
```

This algorithm gives an accuracy of 65.8 percent, which is not bad for a single rule!

Summary

In this chapter, we introduced data mining using Python. If you could run the code in this section (note that the full code is available in the supplied code package), then your computer is set up for much of the rest of the book. Other Python libraries will be introduced in later chapters to perform more specialized tasks.

We used the Jupyter Notebook to run our code, which allows us to immediately view the results of a small section of the code. Jupyter Notebook is a useful tool that will be used throughout the book.

We introduced a simple affinity analysis, finding products that are purchased together. This type of exploratory analysis gives an insight into a business process, an environment, or a scenario. The information from these types of analysis can assist in business processes, find the next big medical breakthrough, or create the next artificial intelligence.

Also, in this chapter, there was a simple classification example using the `OneR` algorithm. This simple algorithm simply finds the best feature and predicts the class that most frequently had this value in the training dataset.

To expand on the outcomes of this chapter, think about how you would implement a variant of `OneR` that can take multiple feature/value pairs into consideration. Take a shot at implementing your new algorithm and evaluating it. Remember to test your algorithm on a separate dataset to the training data. Otherwise, you run the risk of over fitting your data.

Over the next few chapters, we will expand on the concepts of classification and affinity analysis. We will also introduce classifiers in the scikit-learn package and use them to do our machine learning, rather than writing the algorithms ourselves.

2

Classifying with scikit-learn Estimators

The scikit-learn library is a collection of data mining algorithms, written in Python and using a. This library allows users to easily try different algorithms as well as utilize standard tools for doing effective testing and parameter searching. There are many algorithms and utilities in scikit-learn, including many of the commonly used algorithms in modern machine learning.

In this chapter, we focus on setting up a good framework for running data mining procedures. We will use this framework in later chapters, which focus on applications and techniques to use in those situations.

The key concepts introduced in this chapter are as follows:

- **Estimators:** This is to perform classification, clustering, and regression
- **Transformers**: This is to perform pre-processing and data alterations
- **Pipelines**: This is to put together your workflow into a replicable format

scikit-learn estimators

Estimators that allows for the standardized implementation and testing of algorithms a common, lightweight interface for classifiers to follow. By using this interface, we can apply these tools to arbitrary classifiers, without needing to worry about how the algorithms work.

Estimators must have the following two important functions:

- `fit()`: This function performs the training of the algorithm - setting the values of internal parameters. The `fit()` takes two inputs, the training sample dataset and the corresponding classes for those samples.
- `predict()`: This the class of the testing samples that we provide as the only input. This function returns a `NumPy` array with the predictions of each input testing sample.

Most scikit-learn estimators use `NumPy` arrays or a related format for input and output. However this is by convention and not required to use the interface.

There are many estimators implemented in scikit-learn and more in other open source projects that use the same interface. These (SVM), random forests. We will use many of these algorithms in later chapters. In this chapter, we will use the nearest neighbor algorithm.

For this chapter, you will need to install a new library called `matplotlib`. The easiest way to install it is to use `pip3`, as you did in `Chapter 1`, *Getting Started with Data Mining*, to install scikit-learn:

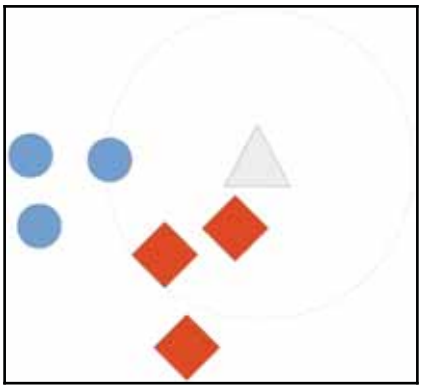
```
$pip3 install matplotlib
```

If you have `matplotlib`, seek the official installation instructions at:
`http://matplotlib.org/users/installing.html`

Nearest neighbors

The **Nearest neighbors** algorithm is our new sample. We take the most similar samples and predict the same class that most of these nearby samples have. This vote is often simply a simple count,although more complicated methods do exist such as weighted voting.

As an example in the below diagram, we wish to predict the class of the triangle, based on which class it is more like (represented here by having similar objects closer together). We seek the three nearest neighbors, which are the two diamonds and one square within the drawn circle. There are more diamonds than circles, and the predicted class for the triangle is, therefore, a diamond:



Nearest neighbors are used for nearly any dataset - however, it can be computationally expensive to compute the distance between all pairs of samples. For example, if there are ten samples in the dataset, there are 45 unique distances to compute. However, if there are 1000 samples, there are nearly 500,000! Various methods exist for improving this speed, such as the use of tree structures for distance computation. Some of these algorithms can be quite complex, but thankfully a version is implemented in scikit-learn already, enabling us to classify on larger datasets. As these tree structures are the default in scikit-learn, we do not need to configure anything to use it.

Nearest neighbors can do poorly in **categorical-based datasets**, with categorical features, and another algorithm should be used for these instead. Nearest Neighbor's issue is due to the difficulty in comparing differences in categorical values, something better left to an algorithm that gives weight to each feature's importance. Comparing categorical features can be done with some distance metrics or pre-processing steps such as one hot encoding that we use in later chapters. Choosing the correct algorithm for the task is one of the difficult issues in data mining, often it can be easiest to test a set of algorithms and see which performs best on your task.

Distance metrics

A key underlying concept in data mining is that of **distance**. If we have two samples, we need to answer questions such as *are these two samples more similar than the other two?* Answering questions like these is important to the outcome of the data mining exercise.

The most common use is **Euclidean** distance, which is the *real-world* distance between two objects. If you were to plot the points on a graph and measure the distance with a ruler, the result would be the Euclidean distance.

A little more formally, the Euclidean distances between points a and b is the square root of the sum of the squared distances for each feature.

Euclidean distance is intuitive but provides poor accuracy if some features have larger values than a value of 0, known as a sparse matrix.

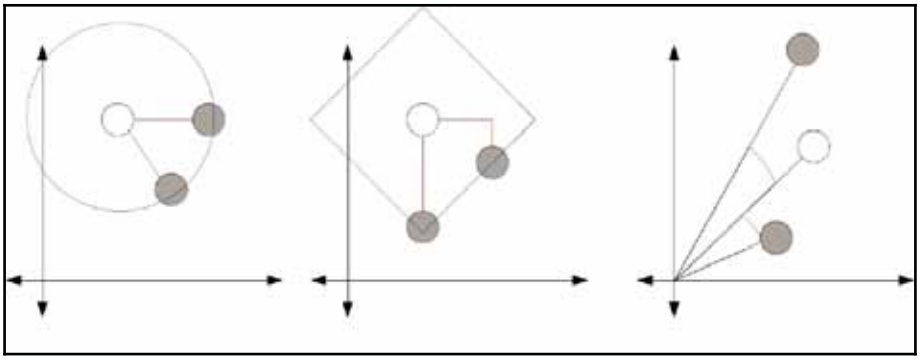
There are other distance metrics in use; two commonly employed ones are the Manhattan and Cosine distance.

The **Manhattan** distance is the sum of the absolute differences in each feature (with no use of square distances).

Intuitively we can imagine Manhattan distance of as the number of moves a Rook piece (also called a Castle) in if it were limited to moving one square at a time. While the Manhattan distance does suffer if some features have larger values than others, the effect is not as dramatic as in the case of Euclidean points if it were limited to moving one square at a time. While the Manhattan distance does suffer if some features have larger values than others, the effect is not as dramatic as in the case of Euclidean.

The **Cosine** distance is better suited to cases where some features are larger than others and when there are lots of zeros in the dataset.

Intuitively, we draw a line from the origin to each of the samples and measure the angle between those lines. We can observe the differences between the algorithms in the following diagram:



In this example, each of the gray circles is exactly the same distance from the white circle. In (a), the distances are Euclidean, and therefore, similar distances fit around a circle. This distance can be measured using a ruler. In (b), the distances are Manhattan, also called City Block. We compute the distance by moving across rows and columns, like how a Rook (Castle) in Chess moves. Finally, in (c), we have the Cosine distance that is measured by computing the angle between the lines drawn from the sample to the vector and ignore the actual length of the line.

The distance metric chosen can have a large impact on the final performance.

For example, if you have many features, the Euclidean distance between random samples converges (due to the famous *curse of dimensionality*). Euclidean distances in high dimension have a hard time comparing samples, as the distances are always nearly the same!

Manhattan distance can be more stable in these circumstances, but if some features have very large values, this can *overrule* lots similarity in other features. For example, if feature A has values between 1 and 2, and another feature B has values between 1000 and 2000, in such a case feature A is unlikely to have any impact on the result. This problem can be addressed through normalization, which makes Manhattan (and Euclidean) distance more reliable with different features, which we will see later in this chapter.

Finally, Cosine distance is a good metric for comparing items with many features, but it discards some information about the length of the vector, which is useful in some applications. We would often use Cosine distance in text mining due to the large number of features inherent in text mining (see `Chapter 6`, *Social Media Insight Using Naive Bayes*).

Ultimately, either a theoretical approach is needed to determine which distance method is needed, or an empirical evaluation is needed to see which performed more effectively. I prefer the empirical approach, but either approach can yield good results.

For this chapter, we will stay with Euclidean distance, using other metrics in later chapters. If you'd like to experiment, then try setting the metric to Manhattan and see how that affects the results.

Loading the dataset

The dataset `Ionosphere`, which high-frequency antennas. The aim of the antennas is to determine whether there is a structure in the ionosphere and a region in the upper atmosphere. We consider readings with a structure to be good, while those that do not have structure are deemed bad. The aim of this application is to build a data mining classifier that can determine whether an image is good or bad.

(Image Credit: https://www.flickr.com/photos/geckzilla/16149273389/)

You can download this dataset for different data mining applications. Go to `http://archive.ics.uci.edu/ml/datasets/Ionosphere` and click on **Data Folder**. Download the `ionosphere.data` and `ionosphere.names` files to a folder on your computer. For this example, I'll assume that you have put the dataset in a directory called `Data` in your `home` folder. You can place the data in another folder, just be sure to update your data folder (here, and in all other chapters).

The location of your home folder depends on your operating system. For Windows, it is usually at `C:Documents` and `Settingsusername`. For Mac or Linux machines, it is usually at `/home/username`. You can get your home folder by running this python code inside a Jupyter Notebook:

```
import os print(os.path.expanduser("~"))
```

For each row in the dataset, there are 35 values. The first 34 are measurements taken from the 17 antennas (two values for each antenna). The last is either 'g' or 'b'; that stands for good and bad, respectively.

Start the Jupyter Notebook server and create a new notebook called **Ionosphere Nearest Neighbors.** To start with, we load up the `NumPy` and `csv` libraries that we will need for our code, and set the data's filename that we will need for our code.

```
import numpy as np
import csv
data_filename = "data/ionosphere.data"
```

We then create the `X` and `y` `NumPy` arrays to store the dataset in. The sizes of these arrays are known from the dataset. Don't worry if you don't know the size of future datasets - we will use other methods to load the dataset in future chapters and you won't need to know this size beforehand:

```
X = np.zeros((351, 34), dtype='float')
y = np.zeros((351,), dtype='bool')
```

The dataset is in a **Comma-Separated Values** (**CSV**) format, which is a commonly used format for datasets. We are going to use the `csv` module to load this file. Import it and set up a `csv` reader object, then loop through the file, setting the appropriate row in `X` and class value in `y` for every line in our dataset:

```
with open(data_filename, 'r') as input_file:
    reader = csv.reader(input_file)
    for i, row in enumerate(reader):
        # Get the data, converting each item to a float
        data = [float(datum) for datum in row[:-1]]
        # Set the appropriate row in our dataset
        X[i] = data
        # 1 if the class is 'g', 0 otherwise
        y[i] = row[-1] == 'g'
```

We now have a dataset of samples and features in `X` as well as the corresponding classes in `y`, as we did in the classification example in `Chapter 1`, Getting Started with Data Mining.

To begin with, try applying the OneR algorithm from `Chapter 1`, *Getting Started with Data Mining* to this dataset. It won't work very well, as the information in this dataset is spread out within the correlations of certain features. OneR is only interested in the values of a single feature and cannot pick up information in more complex datasets very well. Other algorithms, including Nearest Neighbor, merge information from multiple features, making them applicable in more scenarios. The downside is that they are often more computationally expensive to compute.

Moving towards a standard workflow

Estimators scikit-learn have two and `predict()`. We train the algorithm using the `predict()` method on our testing set. We evaluate it using the `predict()` method on our testing set.

1. First, we need to create these training and testing sets. As before, import and run the `train_test_split` function:

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=14)
```

2. Then, we import the `nearest neighbor` class and create an instance for it. We leave the parameters as defaults for now and will test other values later in this chapter. By default, the algorithm will choose the five nearest neighbors to predict the class of a testing sample:

```
from sklearn.neighbors import KNeighborsClassifier estimator =
KNeighborsClassifier()
```

3. After creating our `estimator`, we must then fit it on our training dataset. For the `nearest neighbor` class, this training step simply records our dataset, allowing us to find the nearest neighbor for a new data point, by comparing that point to the training dataset:

```
estimator.fit(X_train, y_train)
```

4. We then train the algorithm with our test set and evaluate with our testing set:

```
y_predicted = estimator.predict(X_test)
accuracy = np.mean(y_test == y_predicted) * 100
print("The accuracy is {0:.1f}%".format(accuracy))
```

This model scores 86.4 percent accuracy, which is impressive for a default algorithm and just a few lines of code! Most scikit-learn default parameters are chosen deliberately to work well with a range of datasets. However, you should always aim to choose parameters based on knowledge of the application experiment. We will use strategies for doing this **parameter search** in later chapters.

Running the algorithm

The previous results are quite good, based on our testing set of data, based on the testing set. However, what happens if we get lucky and choose an easy testing set? Alternatively, what if it was particularly troublesome? We can discard a good model due to poor results resulting from such an *unlucky* split of our data.

The **cross-fold validation** framework is a way to address the problem of choosing a single testing set and is a standard *best-practice* methodology in data mining. The process works by doing many experiments with different training and testing splits, but using each sample in a testing set only once. The procedure is as follows:

1. Split the entire dataset into several sections called folds.
2. For each fold in the data, execute the following steps:
    1. Set that fold aside as the current testing set
    2. Train the algorithm on the remaining folds
    3. Evaluate on the current testing set
3. Report on all the evaluation scores, including the average score.

In this process, each sample is used in the testing set only once, reducing (but not eliminating) the likelihood of choosing lucky testing sets.

Throughout this book, the code examples build upon each other within a chapter. Each chapter's code should be entered into the same Jupyter Notebook unless otherwise specified in-text.

The scikit-learn library contains a few cross-fold validation methods. A `helper` function is given that performs the preceding procedure. We can import it now in our Jupyter Notebook:

```
from sklearn.cross_validation import cross_val_score
```

By `cross_val_score` uses a specific methodology called **Stratified K-Fold** to create folds that have approximately the same proportion of classes in each fold, again reducing the likelihood of choosing poor folds. Stratified K-Fold is a great default -we won't mess with it right now.

Next, we use this new function to evaluate our model using cross-fold validation:

```
scores = cross_val_score(estimator, X, y, scoring='accuracy')
average_accuracy = np.mean(scores) * 100
print("The average accuracy is {0:.1f}%".format(average_accuracy))
```

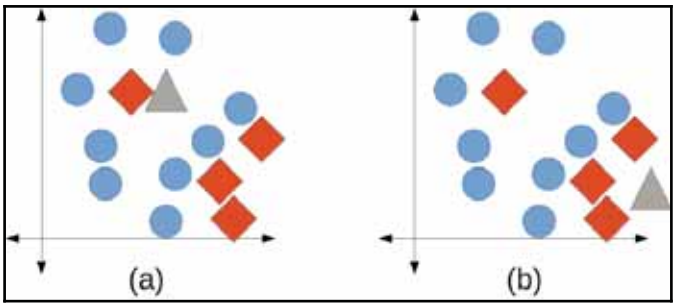
Our new code returns a slightly more modest result of 82.3 percent, but it is still quite good considering we have not yet tried setting better parameters. In the next section, we will see how we would go about changing the parameters to achieve a better outcome.

It is quite natural for variation in results when performing data mining, and attempting to repeat experiments. This is due to variations in how the folds are created and randomness inherent in some classification algorithms. We can deliberately choose to replicate an experiment exactly by setting the random state (which we will do in later chapters). In practice, it's a good idea to rerun experiments multiple times to get a sense of the average result and the spread of the results (the mean and standard deviation) across all experiments.

Setting parameters

Almost all parameters that the user can set, letting algorithms focus more on the specific dataset, rather than only being applicable across a small and specific range of problems. Setting these parameters can be quite difficult, as choosing good parameter values is often highly reliant on features of the dataset.

The nearest neighbor algorithm has several parameters, but the most important one is that of the number of nearest neighbors to use when predicting the class of an unseen attribution. In `-learn`, this parameter is called `n_neighbors`. In the following figure, we show that when this number is too low, a randomly labeled sample can cause an error. In contrast, when it is too high, the actual nearest neighbors have a lower effect on the result:



In figure (a), on the left-hand side, we would usually expect to classify the test sample (the triangle) as a circle. However, if `n_neighbors` is 1, the single red diamond in this area (likely a noisy sample) causes the sample to be predicted as a diamond. In figure (b), on the right-hand side, we would usually expect to classify the test sample as a diamond. However, if `n_neighbors` is 7, the three nearest neighbors (which are all diamonds) are overridden by a large number of circle samples. Nearest neighbors a difficult problem to solve, as the parameter can make a huge difference. Luckily, most of the time the specific parameter value does not greatly affect the end result, and the standard values (usually 5 or 10) are often *near enough*.

With that in mind, we can test out a range of values, and investigate the impact that this parameter has on performance. If we want to test a number of values for the `n_neighbors` parameter, for example, each of the values from 1 to 20, we can rerun the experiment many times by setting `n_neighbors` and observing the result. The code below does this, storing the values in the `avg_scores` and `all_scores` variables.

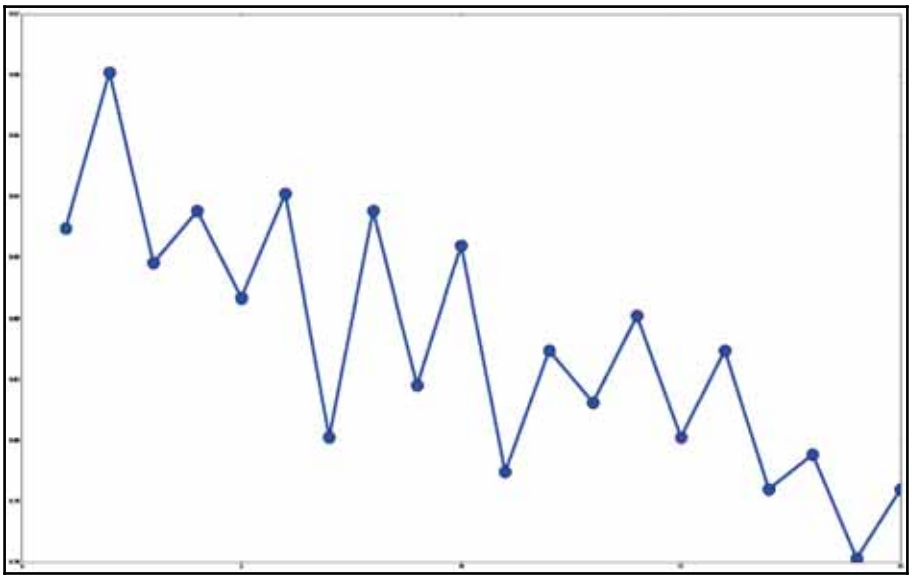
```
avg_scores = []
all_scores = []
parameter_values = list(range(1, 21))  # Include 20
for n_neighbors in parameter_values:
    estimator = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_val_score(estimator, X, y, scoring='accuracy')
avg_scores.append(np.mean(scores))
all_scores.append(scores)
```

We can then plot the relationship between the value of `n_neighbors` and the accuracy. First, we tell the Jupyter Notebook that we want to show plots `inline` in the notebook itself:

```
%matplotlib inline
```

We then import `pyplot` from the `matplotlib` library and plot the parameter values alongside average scores:

```
from matplotlib import pyplot as plt plt.plot(parameter_values,
avg_scores, '-o')
```



While there is a lot of variance, the plot shows a decreasing trend as the number of neighbors increases. With regard to the variance, you can expect large amounts of variance whenever you do evaluations of this nature. To compensate, update the code to run 100 tests, per value of `n_neighbors`.

Preprocessing

When taking measurements of real-world objects, we can often get features in different ranges. For instance, if we measure the qualities of an animal, we might have several features, as follows:

- **Number of legs**: This is between the range of 0-8 for most animals, while some have more! more! more!
- **Weight**: This is between the ranges of only a few micrograms, all the way to a blue whale with a weight of 190,000 kilograms!

- **Number of hearts**: This can be between zero to five, in the case of the earthworm.

For a mathematical-based algorithm to compare each of these features, the differences in the scale, range, and units can be difficult to interpret. If we used the above features in many algorithms, the weight would probably be the most influential feature due to only the larger numbers and not anything to do with the actual effectiveness of the feature.

One of the possible strategies *normalizes* the features so that they all have the same range, or the values are turned into categories like *small*, *medium* and *large*. Suddenly, the large differences in the types of features have less of an impact on the algorithm and can lead to large increases in the accuracy.

Pre-processing can also be used to choose only the more effective features, create new features, and so on. Pre-processing in scikit-learn is done through `Transformer` objects, which take a dataset in one form and return an altered dataset after some transformation of the data. These don't have to be numerical, as Transformers are also used to extract features-however, in this section, we will stick with pre-processing.

We can show an example of the problem by *breaking* the `Ionosphere` dataset. While this is only an example, many real-world datasets have problems of this form.

1. First, we create a copy of the array so that we do not alter the original dataset:

```
X_broken = np.array(X)
```

2. Next, we *break* the dataset by dividing every second feature by `10`:

```
X_broken[:,::2] /= 10
```

In theory, this should not have a great effect on the result. After all, the values of these features are still relatively the same. The major issue is that the scale has changed and the odd features are now *larger* than the even features. We can see the effect of this by computing the accuracy:

```
estimator = KNeighborsClassifier()
original_scores = cross_val_score(estimator, X, y,scoring='accuracy')
print("The original average accuracy for is
{0:.1f}%".format(np.mean(original_scores) * 100))
broken_scores = cross_val_score(estimator, X_broken, y,
scoring='accuracy')
print("The 'broken' average accuracy for is
{0:.1f}%".format(np.mean(broken_scores) * 100))
```

This testing methodology gives a score of 82.3 percent for the original dataset, which drops down to 71.5 percent on the broken dataset. We can fix this by scaling all the features to the range `0` to `1`.

Standard pre-processing

The pre-processing we will perform for this experiment is called feature-based normalization, which we perform using scikit-learn's `MinMaxScaler` class. Continuing with the Jupyter Notebook from the rest of this chapter, first, we import this class:

```
fromsklearn.preprocessing import MinMaxScaler
```

This class takes each feature and scales it to the range `0` to `1`. This pre-processor replaces the minimum value with `0`, the maximum with `1`, and the other values somewhere in between based on a linear mapping.

To apply our pre-processor, we run the `transform` function on it. Transformers often need to be trained first, in the same way that the classifiers do. We can combine these steps by running the `fit_transform` function instead:

```
X_transformed = MinMaxScaler().fit_transform(X)
```

Here, `X_transformed` will have the same shape as *X*. However, each column will have a maximum of `1` and a minimum of `0`.

There are various other forms of normalizing in this way, which is effective for other applications and feature types:

- Ensure the sum of the values for each sample equals to 1, using `sklearn.preprocessing.Normalizer`
- Force each feature to have a zero mean and a variance of 1, using `sklearn.preprocessing.StandardScaler`, which is a commonly used starting point for normalization
- Turn numerical features into binary features, where any value above a threshold is 1 and any below is 0, using `sklearn.preprocessing.Binarizer`

We will use combinations of these pre-processors in later chapters, along with other types of `Transformers` object.

Pre-processing is a critical step in the data mining pipeline and one that can mean the difference between a bad and great result.

Putting it all together

We can now create a workflow by combining the code from the previous sections, using the broken dataset previously calculated:

```
X_transformed = MinMaxScaler().fit_transform(X_broken)
estimator = KNeighborsClassifier()
transformed_scores = cross_val_score(estimator, X_transformed, y,
scoring='accuracy')
print("The average accuracy for is
{0:.1f}%".format(np.mean(transformed_scores) * 100))
```

We now recover our original score of 82.3 percent accuracy. The `MinMaxScaler` resulted in features of the same scale, meaning that no features overpowered others by simply being bigger values. While the Nearest Neighbor algorithm can be confused with larger features, some algorithms handle scale differences better. In contrast, some are much worse!

Pipelines

As experiments grow, so does the complexity of the operations. We may split up our dataset, binarize features, perform feature-based scaling, perform sample-based scaling, and many more operations.

Keeping track of these operations can get quite confusing and can result in being unable to replicate the result. Problems include forgetting a step, incorrectly applying a transformation, or adding a transformation that wasn't needed.

Another issue is the order of the code. In the previous section, we created our `X_transformed` dataset and then created a new estimator for the cross validation.If we had multiple steps, we would need to track these changes to the dataset in code.

Pipelines are a construct that addresses these problems (and others, which we will see in the next chapter). Pipelines store the steps in your data mining workflow. They can take your raw data in, perform all the necessary transformations, and then create a prediction. This allows us to use pipelines in functions such as `cross_val_score`, where they expect an estimator. First, import the `Pipeline` object:

```
fromsklearn.pipeline import Pipeline
```

Pipelines take a list of steps as input, representing the chain of the data mining application. The last step needs to be an Estimator, while all previous steps are Transformers. The input dataset is altered by each Transformer, with the output of one step being the input of the next step. Finally, we classify the samples by the last step's estimator. In our pipeline, we have two steps:

1. Use `MinMaxScaler` to scale the feature values from 0 to 1
2. Use `KNeighborsClassifier` as the classification algorithms

We then represent each step using a tuple `('name', step)`. We can then create our pipeline:

```
scaling_pipeline = Pipeline([('scale', MinMaxScaler()),
                             ('predict', KNeighborsClassifier())])
```

The key here is the list of tuples. The first tuple is our scaling step and the second tuple is the predicting step. We give each step a name: the first we call `scale` and the second we call `predict`, but you can choose your own names. The second part of the tuple is the actual `Transformer` or `estimator` object.

Running this pipeline is now very easy, using the cross-validation code from before:

```
scores = cross_val_score(scaling_pipeline, X_broken, y, scoring='accuracy')
print("The pipeline scored an average accuracy for is
{0:.1f}%".format(np.mean(transformed_scores) * 100))
```

This gives us the same score as before (82.3 percent), which is expected, as we are running exactly the same steps, just with an improved interface.

In later chapters, we will use more advanced testing methods and setting up pipelines is a great way to ensure that the code complexity does not grow unmanageably.

Summary

In this chapter, we used several of scikit-learn's methods for building a standard workflow to run and evaluate data mining models. We introduced the Nearest Neighbors algorithm, which is implemented in scikit-learn as an estimator. Using this class is quite easy; first, we call the `fit` function on our training data, and second, we use the `predict` function to predict the class of testing samples.

We then looked at pre-processing by fixing poor feature scaling. This was done using a `Transformer` object and the `MinMaxScaler` class. These functions also have a `fit` method and then a transform, which takes data of one form as an input and returns a transformed dataset as an output.

To investigate these transformations further, try swapping out the `MinMaxScaler` with some of the other mentioned transformers. Which is the most effective and why would this be the case?

Other transformers also exist in scikit-learn, which we will use later in this book, such as PCA. Try some of these out as well, referencing scikit-learn's excellent documentation at `https://scikit-learn.org/stable/modules/preprocessing.html`

In the next chapter, we will use these concepts in a larger example, predicting the outcome of sports matches using real-world data.

3

Predicting Sports Winners with Decision Trees

In this chapter, we will look at predicting the winner of sports matches using a different type of classification algorithm to the ones we have seen so far: **decision trees**. These algorithms have a number of advantages over other algorithms. One of the main advantages is that they are readable by humans, allowing for their use in human-driven decision making. In this way, decision trees can be used to learn a procedure, which could then be given to a human to perform if needed. Another advantage is that they work with a variety of features, including categorical, which we will see in this chapter.

We will cover the following topics in this chapter:

- Using the pandas library for loading and manipulating data
- Decision trees for classification
- Random forests to improve upon decision trees
- Using real-world datasets in data mining
- Creating new features and testing them in a robust framework

Loading the dataset

In this chapter, we will look at predicting the winner of games of the **National Basketball Association** (**NBA**). Matches in the NBA are often close and can be decided at the last minute, making predicting the winner quite difficult. Many sports share this characteristic, whereby the (generally) better team could be beaten by another team on the right day.

Various research into predicting the winner suggests that there may be an upper limit to sports outcome prediction accuracy which, depending on the sport, is between 70 percent and 80 percent. There is a significant amount of research being performed into sports prediction, often through data mining or statistics-based methods.

In this chapter, we are going to have a look at an entry level basketball match prediction algorithm, using decision trees for determining whether a team will win a given match. Unfortunately, it doesn't quite make as much profit as the models that sports betting agencies use, which are often a bit more advanced, more complex, and ultimately, more accurate.

Collecting the data

The data we will be using is the match history data for the NBA for the 2015-2016 season. The website `http://basketball-reference.com` contains a significant number of resources and statistics collected from the NBA and other leagues. To download the dataset, perform the following steps:

1. Navigate to `http://www.basketball-reference.com/leagues/NBA_2016_games.html` in your web browser.
2. Click **Share & more.**
3. Click **Get table as CSV (for Excel).**
4. Copy the data, including the heading, into a text file named `basketball.csv`.
5. Repeat this process for the other months, except do not copy the heading.

This will give you a CSV file containing the results from each game of this season of the NBA. Your file should contain 1316 games and a total of 1317 lines in the file, including the header line.

CSV files are text files where each line contains a new row and each value is separated by a comma (hence the name). CSV files can be created manually by typing into a text editor and saving with a `.csv` extension. They can be opened in any program that can read text files but can also be opened in Excel as a spreadsheet. Excel (and other spreadsheet programs) can usually convert a spreadsheet to CSV as well.

We will load the file with the `pandas` library, which is an incredibly useful library for manipulating data. Python also contains a built-in library called `csv` that supports reading and writing CSV files. However, we will use pandas, which provides more powerful functions that we will use later in the chapter for creating new features.

For this chapter, you will need to install pandas. The easiest way to install it is to use Anaconda's `conda` installer, as you did in `Chapter 1`, *Getting Started with data mining to install scikit-learn*:

```
$ conda install pandas
```

If you have difficulty in installing pandas, head to the project's website at `http://pandas.pydata.org/getpandas.html` and read the installation instructions for your system.

Using pandas to load the dataset

The `pandas` library is a library for loading, managing, and manipulating data. It handles data structures behind-the-scenes and supports data analysis functions, such as computing the mean and grouping data by value.

When doing multiple data mining experiments, you will find that you write many of the same functions again and again, such as reading files and extracting features. Each time this reimplementation happens, you run the risk of introducing bugs. Using a high-quality library such as `pandas` significantly reduces the amount of work needed to do these functions, and also gives you more confidence in using well-tested code to underly your own programs.

Throughout this book, we will be using pandas a lot, introducing use cases as we go and new functions as needed.

We can load the dataset using the `read_csv` function:

```
import pandas as pd
data_filename = "basketball.csv"
dataset = pd.read_csv(data_filename)
```

The result of this is a pandas **DataFrame**, and it has some useful functions that we will use later on. Looking at the resulting dataset, we can see some issues. Type the following and run the code to see the first five rows of the dataset:

```
dataset.head(5)
```

Here's the output:

```
In [2]: dataset.head()
```

Out[2]:

	Date	Start (ET)	Visitor/Neutral	PTS	Home/Neutral	PTS.1	Unnamed: 6	Unnamed: 7	Notes
0	Tue Oct 27 2015	8:00 pm	Detroit Pistons	106	Atlanta Hawks	94	Box Score	NaN	NaN
1	Tue Oct 27 2015	8:00 pm	Cleveland Cavaliers	95	Chicago Bulls	97	Box Score	NaN	NaN
2	Tue Oct 27 2015	10:30 pm	New Orleans Pelicans	95	Golden State Warriors	111	Box Score	NaN	NaN
3	Wed Oct 28 2015	7:30 pm	Philadelphia 76ers	95	Boston Celtics	112	Box Score	NaN	NaN
4	Wed Oct 28 2015	7:30 pm	Chicago Bulls	115	Brooklyn Nets	100	Box Score	NaN	NaN

Just reading the data with no parameters resulted in quite a usable dataset, but it has some issues which we will address in the next section.

Cleaning up the dataset

After looking at the output, we can see a number of problems:

- The date is just a string and not a date object
- From visually inspecting the results, the headings aren't complete or correct

These issues come from the data and we could fix this by altering the data itself. However, in doing this, we could forget the steps we took or misapply them; that is, we can't replicate our results. As with the previous section where we used pipelines to track the transformations we made to a dataset, we will use pandas to apply transformations to the raw data itself.

The `pandas.read_csv` function has parameters to fix each of these issues, which we can specify when loading the file. We can also change the headings after loading the file, as shown in the following code:

```
dataset = pd.read_csv(data_filename, parse_dates=["Date"]) dataset.columns
        = ["Date", "Start (ET)", "Visitor Team", "VisitorPts",
           "Home Team", "HomePts", "OT?", "Score Type", "Notes"]
```

The results have significantly improved, as we can see if we print out the resulting data frame:

```
dataset.head()
```

The output is as follows:

```
In [4]: dataset.head()
```

Out[4]:

	Date	Start (ET)	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Score Type	Notes
0	2015-10-27	8:00 pm	Detroit Pistons	106	Atlanta Hawks	94	Box Score	NaN	NaN
1	2015-10-27	8:00 pm	Cleveland Cavaliers	95	Chicago Bulls	97	Box Score	NaN	NaN
2	2015-10-27	10:30 pm	New Orleans Pelicans	95	Golden State Warriors	111	Box Score	NaN	NaN
3	2015-10-28	7:30 pm	Philadelphia 76ers	95	Boston Celtics	112	Box Score	NaN	NaN
4	2015-10-28	7:30 pm	Chicago Bulls	115	Brooklyn Nets	100	Box Score	NaN	NaN

Even in well-compiled data sources such as this one, you need to make some adjustments. Different systems have different nuances, resulting in data files that are not quite compatible with each other. When loading a dataset for the first time, always check the data loaded (even if it's a known format) and also check the data types of the data. In pandas, this can be done with the following code:

```
print(dataset.dtypes)
```

Now that we have our dataset in a consistent format, we can compute a **baseline**, which is an easy way to get a good accuracy on a given problem. Any decent data mining solution should beat this baseline figure.

For a product recommendation system, a good baseline is to simply *recommend the most popular product*.
For a classification task, it can be to *always predict the most frequent task*, or alternatively applying a very simple classification algorithm like **OneR**.

For our dataset, each match has two teams: a home team and a visitor team. An obvious baseline for this task is 50 percent, which is our expected accuracy if we simply guessed a winner at random. In other words, choosing the predicted winning team randomly will (over time) result in an accuracy of around 50 percent. With a little domain knowledge, however, we can use a better baseline for this task, which we will see in the next section.

Extracting new features

We will now extract some features from this dataset by combining and comparing the existing data. First, we need to specify our class value, which will give our classification algorithm something to compare against to see if its prediction is correct or not. This could be encoded in a number of ways; however, for this application, we will specify our class as 1 if the home team wins and 0 if the visitor team wins. In basketball, the team with the most points wins. So, while the data set doesn't specify who wins directly, we can easily compute it.

We can specify the data set by the following:

```
dataset["HomeWin"] = dataset["VisitorPts"] < dataset["HomePts"]
```

We then copy those values into a NumPy array to use later for our scikit-learn classifiers. There is not currently a clean integration between pandas and scikit-learn, but they work nicely together through the use of NumPy arrays. While we will use pandas to extract features, we will need to extract the values to use them with scikit-learn:

```
y_true = dataset["HomeWin"].values
```

The preceding array now holds our class values in a format that scikit-learn can read.

By the way, the better baseline figure for sports prediction is to predict the home team in every game. Home teams are shown to have an advantage in nearly all sports across the world. How big is this advantage? Let's have a look:

```
dataset["HomeWin"].mean()
```

The resulting value, around 0.59, indicates that the home team wins 59 percent of games on average. This is higher than 50 percent from random chance and is a simple rule that applies to most sports.

We can also start creating some features to use in our data mining for the input values (the `X` array). While sometimes we can just throw the raw data into our classifier, we often need to derive continuous numerical or categorical features from our data.

For our current dataset, we can't really use the features already present (in their current form) to do a prediction. We wouldn't know the scores of a game before we would need to predict the outcome of the game, so we can not use them as features. While this might sound obvious, it can be easy to miss.

The first two features we want to create to help us predict which team will win are whether either of those two teams won their previous game. This would roughly approximate which team is currently playing well.

We will compute this feature by iterating through the rows in order and recording which team won. When we get to a new row, we look up whether the team won the last time we saw them.

We first create a (default) dictionary to store the team's last result:

```
from collections import defaultdict
won_last = defaultdict(int)
```

We then create a new feature on our dataset to store the results of our new features:

```
dataset["HomeLastWin"] = 0
dataset["VisitorLastWin"] = 0
```

The key of this dictionary will be the team and the value will be whether they won their previous game. We can then iterate over all the rows and update the current row with the team's last result:

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    row["HomeLastWin"] = won_last[home_team]
    dataset.set_value(index, "HomeLastWin", won_last[home_team])
    dataset.set_value(index, "VisitorLastWin", won_last[visitor_team])
    won_last[home_team] = int(row["HomeWin"])
    won_last[visitor_team] = 1 - int(row["HomeWin"])
```

Note that the preceding code relies on our dataset being in chronological order. Our dataset is in order; however, if you are using a dataset that is not in order, you will need to replace `dataset.iterrows()` with `dataset.sort("Date").iterrows()`.

Those last two lines in the loop update our dictionary with either a 1 or a 0, depending on which team won the *current* game. This information is used for the next game each team plays.

After the preceding code runs, we will have two new features: `HomeLastWin` and `VisitorLastWin`. Have a look at the dataset using `dataset.head(6)` to see an example of a home team and a visitor team that won their recent game. Have a look at other parts of the dataset using the panda's indexer:

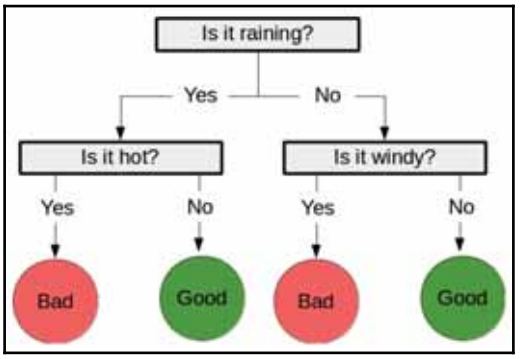
```
dataset.ix[1000:1005]
```

Currently, this gives a false value to all teams (including the previous year's champion!) when they are first seen. We could improve this feature using the previous year's data, but we will not do that in this chapter.

Decision trees

Decision trees are a class of supervised learning algorithms like a flow chart that consists of a sequence of nodes, where the values for a sample are used to make a decision on the next node to go to.

The following example gives a very good idea of how decision trees are a class of supervised learning algorithms:



As with most classification algorithms, there are two stages to using them:

- The first stage is the **training** stage, where a tree is built using training data. While the nearest neighbor algorithm from the previous chapter did not have a training phase, it is needed for decision trees. In this way, the nearest neighbor algorithm is a lazy learner, only doing any work when it needs to make a prediction. In contrast, decision trees, like most classification methods, are eager learners, undertaking work at the training stage and therefore needing to do less in the predicting stage.

- The second stage is the **predicting** stage, where the trained tree is used to predict the classification of new samples. Using the previous example tree, a data point of `["is raining", "very windy"]` would be classed as *bad weather*.

There are many algorithms for creating decision trees. Many of these algorithms are iterative. They start at the base node and decide the best feature to use for the first decision, then go to each node and choose the next best feature, and so on. This process is stopped at a certain point when it is decided that nothing more can be gained from extending the tree further.

The `scikit-learn` package implements the **Classification and Regression Trees** (**CART**) algorithm as its default dDecision tree class, which can use both categorical and continuous features.

Parameters in decision trees

One of the most important parameters for a Decision Tree is the **stopping criterion**. When the tree building is nearly completed, the final few decisions can often be somewhat arbitrary and rely on only a small number of samples to make their decision. Using such specific nodes can result in trees that significantly overfit the training data. Instead, a stopping criterion can be used to ensure that the Decision Tree does not reach this exactness.

Instead of using a stopping criterion, the tree could be created in full and then trimmed. This trimming process removes nodes that do not provide much information to the overall process. This is known as **pruning** and results in a model that generally does better on new datasets because it hasn't overfitted the training data.

The decision tree implementation in scikit-learn provides a method to stop the building of a tree using the following options:

- **min_samples_split**: This specifies how many samples are needed in order to create a new node in the Decision Tree
- **min_samples_leaf**: This specifies how many samples must be resulting from a node for it to stay

The first dictates whether a decision node will be created, while the second dictates whether a decision node will be kept.

Another parameter for decision trees is the criterion for creating a decision. **Gini impurity** and **information gain** are two popular options for this parameter:

- **Gini impurity**: This is a measure of how often a decision node would incorrectly predict a sample's class
- **Information gain**: This uses information-theory-based entropy to indicate how much extra information is gained by the decision node

These parameter values do approximately the same thing--decide which rule and value to use to split a node into subnodes. The value itself is simply which metric to use to determine that split, however this can make a significant impact on the final models.

Using decision trees

We can import the `DecisionTreeClassifier` class and create a Decision Tree using scikit-learn:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=14)
```

We used 14 for our `random_state` again and will do so for most of the book. Using the same random seed allows for replication of experiments. However, with your experiments, you should mix up the random state to ensure that the algorithm's performance is not tied to the specific value.

We now need to extract the dataset from our pandas data frame in order to use it with our `scikit-learn` classifier. We do this by specifying the columns we wish to use and using the values parameter of a view of the data frame. The following code creates a dataset using our last win values for both the home team and the visitor team:

```
X_previouswins = dataset[["HomeLastWin", "VisitorLastWin"]].values
```

Decision trees are estimators, as introduced in `Chapter 2`, *Classifying using scikit-learn Estimators*, and therefore have `fit` and `predict` methods. We can also use the `cross_val_score` method to get the average score (as we did previously):

```
from sklearn.cross_validation import cross_val_score
import numpy as np
scores = cross_val_score(clf, X_previouswins, y_true,
scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores 59.4 percent: we are better than choosing randomly! However, we aren't beating our other baseline of just choosing the home team. In fact, we are pretty much exactly the same. We should be able to do better. **Feature engineering** is one of the most difficult tasks in data mining, and choosing *good* **features** is key to getting good outcomes—more so than choosing the right algorithm!

Sports outcome prediction

We may be able to do better by trying other features. We have a method for testing how accurate our models are. The `cross_val_score` method allows us to try new features.

There are many possible features we could use, but we will try the following questions:

- Which team is considered better generally?
- Which team won their last encounter?

We will also try putting the raw teams into the algorithm, to check whether the algorithm can learn a model that checks how different teams play against each other.

Putting it all together

For the first feature, we will create a feature that tells us if the home team is generally better than the visitors. To do this, we will load the standings (also called a ladder in some sports) from the NBA in the previous season. A team will be considered better if it ranked higher in 2015 than the other team.

To obtain the standings data, perform the following steps:

1. Navigate to `http://www.basketball-reference.com/leagues/NBA_2015_standings.html` in your web browser.
2. Select **Expanded Standings** to get a single list for the entire league.
3. Click on the **Export link**.
4. Copy the text and save it in a text/CSV file called `standings.csv` in your data folder.

Back in your Jupyter Notebook, enter the following lines into a new cell. You'll need to ensure that the file was saved into the location pointed to by the data_folder variable. The code is as follows:

```
import os
standings_filename = os.path.join(data_folder, "standings.csv")
standings = pd.read_csv(standings_filename, skiprows=1)
```

You can view the ladder by just typing standings into a new cell and running the code:

```
standings.head()
```

The output is as follows:

```
In [20]: standings.head()
```

Out[20]:

	Rk	Team	Overall	Home	Road	E	W	A	C	SE	...	Post	≤3	≥10	Oct	Nov	Dec	Jan	Feb	Mar	Apr
0	1	Golden State Warriors	67-15	39-2	28-13	25-5	42-10	9-1	7-3	9-1	...	25-6	5-3	45-9	1-0	13-2	11-3	12-3	8-3	16-2	6-2
1	2	Atlanta Hawks	60-22	35-6	25-16	38-14	22-8	12-6	14-4	12-4	...	17-11	6-4	30-10	0-1	9-5	14-2	17-0	7-4	9-7	4-3
2	3	Houston Rockets	56-26	30-11	26-15	23-7	33-19	9-1	8-2	6-4	...	20-9	8-4	31-14	2-0	11-4	9-5	11-6	7-3	10-6	6-2
3	4	Los Angeles Clippers	56-26	30-11	26-15	19-11	37-15	7-3	6-4	6-4	...	21-7	3-5	33-9	2-0	9-5	11-6	11-4	5-6	11-5	7-0
4	5	Memphis Grizzlies	55-27	31-10	24-17	20-10	35-17	8-2	5-5	7-3	...	16-13	9-3	26-13	2-0	13-2	8-6	12-4	7-4	9-8	4-3

Next, we create a new feature using a similar pattern to the previous feature. We iterate over the rows, looking up the standings for the home team and visitor team. The code is as follows:

```
dataset["HomeTeamRanksHigher"] = 0
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    home_rank = standings[standings["Team"] == home_team]["Rk"].values[0]
    visitor_rank = standings[standings["Team"] ==
visitor_team]["Rk"].values[0]
    row["HomeTeamRanksHigher"] = int(home_rank > visitor_rank)
    dataset.set_value(index, "HomeTeamRanksHigher", int(home_rank <
visitor_rank))
```

Next, we use the `cross_val_score` function to test the result. First, we extract the dataset:

```
X_homehigher = dataset[["HomeLastWin", "VisitorLastWin",
"HomeTeamRanksHigher"]].values
```

Then, we create a new `DecisionTreeClassifier` and run the evaluation:

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_homehigher, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This now scores 60.9 percent even better than our previous result, and now better than just choosing the home team every time. Can we do better?

Next, let's test which of the two teams won their last match against each other. While rankings can give some hints on who won (the higher ranked team is more likely to win), sometimes teams play better against other teams. There are many reasons for this--for example, some teams may have strategies or players that work against specific teams really well. Following our previous pattern, we create a dictionary to store the winner of the past game and create a new feature in our data frame. The code is as follows:

```
last_match_winner = defaultdict(int)
dataset["HomeTeamWonLast"] = 0

for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    teams = tuple(sorted([home_team, visitor_team])) # Sort for a
consistent ordering
    # Set in the row, who won the last encounter
    home_team_won_last = 1 if last_match_winner[teams] == row["Home Team"]
else 0
    dataset.set_value(index, "HomeTeamWonLast", home_team_won_last)
    # Who won this one?
    winner = row["Home Team"] if row["HomeWin"] else row["Visitor Team"]
    last_match_winner[teams] = winner
```

This feature works much like our previous rank-based feature. However, instead of looking up the ranks, this features creates a tuple called `teams`, and then stores the previous result in a dictionary. When those two teams play each other next, it recreates this tuple, and looks up the previous result. Our code doesn't differentiate between home games and visitor games, which might be a useful improvement to look at implementing.

Next, we need to evaluate. The process is pretty similar to before, except we add the new feature into the extracted values:

```
X_lastwinner = dataset[[ "HomeTeamWonLast", "HomeTeamRanksHigher",
"HomeLastWin", "VisitorLastWin",]].values
clf = DecisionTreeClassifier(random_state=14, criterion="entropy")

scores = cross_val_score(clf, X_lastwinner, y_true, scoring='accuracy')
```

```
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores 62.2 percent. Our results are getting better and better.

Finally, we will check what happens if we throw a lot of data at the Decision Tree, and see if it can learn an effective model anyway. We will enter the teams into the tree and check whether a Decision Tree can learn to incorporate that information.

While decision trees are capable of learning from categorical features, the implementation in `scikit-learn` requires those features to be encoded as numbers and features, instead of string values. We can use the `LabelEncoder` **transformer** to convert the string-based team names into assigned integer values. The code is as follows:

```
from sklearn.preprocessing import LabelEncoder
encoding = LabelEncoder()
encoding.fit(dataset["Home Team"].values)
home_teams = encoding.transform(dataset["Home Team"].values)
visitor_teams = encoding.transform(dataset["Visitor Team"].values)
X_teams = np.vstack([home_teams, visitor_teams]).T
```

We should use the same transformer for encoding both the home team and visitor teams. This is so that the same team gets the same integer value as both a home team and visitor team. While this is not critical to the performance of this application, it is important and failing to do this may degrade the performance of future models.

These integers can be fed into the Decision Tree, but they will still be interpreted as continuous features by `DecisionTreeClassifier`. For example, teams may be allocated integers from 0 to 16. The algorithm will see teams 1 and 2 as being similar, while teams 4 and 10 will be very different--but this makes no sense as all. All of the teams are different from each other--two teams are either the same or they are not!

To fix this inconsistency, we use the `OneHotEncoder` **transformer** to encode these integers into a number of binary features. Each binary feature will be a single value for the feature. For example, if the NBA team Chicago Bulls is allocated as integer 7 by the `LabelEncoder`, then the seventh feature returned by the `OneHotEncoder` will be a 1 if the team is Chicago Bulls and 0 for all other features/teams. This is done for every possible value, resulting in a much larger dataset. The code is as follows:

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
X_teams = onehot.fit_transform(X_teams).todense()
```

Next, we run the Decision Tree as before on the new dataset:

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_teams, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This scores an accuracy of 62.8 percent. The score is better still, even though the information given is just the teams playing. It is possible that the larger number of features were not handled properly by the decision trees. For this reason, we will try changing the algorithm and see if that helps. Data mining can be an iterative process of trying new algorithms and features.

Random forests

A single Decision Tree can learn quite complex functions. However, decision trees are prone to overfitting--learning rules that work only for the specific training set and don't generalize well to new data.

One of the ways that we can adjust for this is to limit the number of rules that it learns. For instance, we could limit the depth of the tree to just three layers. Such a tree will learn the best rules for splitting the dataset at a global level, but won't learn highly specific rules that separate the dataset into highly accurate groups. This trade-off results in trees that may have a good generalization, but an overall slightly poorer performance on the training dataset.

To compensate for this, we could create many of these *limited* decision trees and then ask each to predict the class value. We could take a majority vote and use that answer as our overall prediction. Random Forests is an algorithm developed from this insight.

There are two problems with the aforementioned procedure. The first problem is that building decision trees is largely deterministic—using the same input will result in the same output each time. We only have one training dataset, which means our input (and therefore the output) will be the same if we try to build multiple trees. We can address this by choosing a random subsample of our dataset, effectively creating new training sets. This process is called **bagging** and it can be very effective in many situations in data mining.

The second problem we might run into with creating many decision trees from similar data is that the features that are used for the first few decision nodes in our tree will tend to be similar. Even if we choose random subsamples of our training data, it is still quite possible that the decision trees built will be largely the same. To compensate for this, we also choose a random subset of the features to perform our data splits on.

Then, we have randomly built trees using randomly chosen samples, using (nearly) randomly chosen features. This is a random forest and, perhaps unintuitively, this algorithm is very effective for many datasets, with little need to tune many parameters of the model.

How do ensembles work?

The randomness inherent in random forests may make it seem like we are leaving the results of the algorithm up to chance. However, we apply the benefits of averaging to nearly randomly built decision trees, resulting in an algorithm that reduces the variance of the result.

Variance is the error introduced by variations in the training dataset on the algorithm. Algorithms with a high variance (such as decision trees) can be greatly affected by variations to the training dataset. This results in models that have the problem of overfitting. In contrast, **bias** is the error introduced by assumptions in the algorithm rather than anything to do with the dataset, that is, if we had an algorithm that presumed that all features would be normally distributed, then our algorithm may have a high error if the features were not.

Negative impacts from bias can be reduced by analyzing the data to see if the classifier's data model matches that of the actual data.

To use an extreme example, a classifier that always predicts true, regardless of the input, has a very high bias. A classifier that always predicts randomly would have a very high variance. Each classifier has a high degree of error but of a different nature.

By averaging a large number of decision trees, this variance is greatly reduced. This results, at least normally, in a model with a higher overall accuracy and better predictive power. The trade-offs are an increase in time and an increase in the bias of the algorithm.

In general, ensembles work on the assumption that errors in prediction are effectively random and that those errors are quite different from one classifier to another. By averaging the results across many models, these random errors are canceled out—leaving the true prediction. We will see many more ensembles in action throughout the rest of the book.

Setting parameters in Random Forests

The Random Forest implementation in scikit-learn is called `RandomForestClassifier`, and it has a number of parameters. As Random Forests use many instances of `DecisionTreeClassifier`, they share many of the same parameters such as the `criterion` (Gini Impurity or Entropy/information gain), `max_features`, and `min_samples_split`.

There are some new parameters that are used in the ensemble process:

- `n_estimators`: This dictates how many decision trees should be built. A higher value will take longer to run, but will (probably) result in a higher accuracy.
- `oob_score`: If true, the method is tested using samples that aren't in the random subsamples chosen for training the decision trees.
- `n_jobs`: This specifies the number of cores to use when training the decision trees in parallel.

The `scikit-learn` package uses a library called **Joblib** for inbuilt parallelization. This parameter dictates how many cores to use. By default, only a single core is used--if you have more cores, you can increase this, or set it to -1 to use all cores.

Applying random forests

Random forests in scikit-learn use the **Estimator** interface, allowing us to use almost the exact same code as before to do cross-fold validation:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_teams, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This results in an immediate benefit of 65.3 percent, up by 2.5 points by just swapping the classifier.

Random forests, using subsets of the features, should be able to learn more effectively with more features than normal decision trees. We can test this by throwing more features at the algorithm and seeing how it goes:

```
X_all = np.hstack([X_lastwinner, X_teams])
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_all, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

This results in 63.3 percent—a drop in performance! One cause is the randomness inherent in random forests only chose some features to use rather than others. Further, there are many more features in `X_teams` than in `X_lastwinner`, and having the extra features results in less relevant information being used. That said, don't get too excited by small changes in percentages, either up or down. Changing the random state value will have more of an impact on the accuracy than the slight difference between these feature sets that we just observed. Instead, you should run many tests with different random states, to get a good sense of the mean and spread of accuracy values.

We can also try some other parameters using the `GridSearchCV` class, as we introduced in `Chapter 2`, *Classifying using scikit-learn Estimators*:

```
from sklearn.grid_search import GridSearchCV

parameter_space = {
 "max_features": [2, 10, 'auto'],
 "n_estimators": [100, 200],
 "criterion": ["gini", "entropy"],
 "min_samples_leaf": [2, 4, 6],
}

clf = RandomForestClassifier(random_state=14)
grid = GridSearchCV(clf, parameter_space)
grid.fit(X_all, y_true)
print("Accuracy: {0:.1f}%".format(grid.best_score_ * 100))
```

This has a much better accuracy of 67.4 percent!

If we wanted to see the parameters used, we can print out the best model that was found in the grid search. The code is as follows:

```
print(grid.best_estimator_)
```

The result shows the parameters that were used in the best scoring model:

```
RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='entropy',
            max_depth=None, max_features=2, max_leaf_nodes=None,
            min_samples_leaf=2, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
            oob_score=False, random_state=14, verbose=0, warm_start=False)
```

Engineering new features

In the previous few examples, we saw that changing the features can have quite a large impact on the performance of the algorithm. Through our small amount of testing, we had more than 10 percent variance just from the features.

You can create features that come from a simple function in pandas by doing something like this:

```
dataset["New Feature"] = feature_creator()
```

The feature_creator function must return a list of the feature's value for each sample in the dataset. A common pattern is to use the dataset as a parameter:

```
dataset["New Feature"] = feature_creator(dataset)
```

You can create those features more directly by setting all the values to a single default value, like 0 in the next line:

```
dataset["My New Feature"] = 0
```

You can then iterate over the dataset, computing the features as you go. We used this format in this chapter to create many of our features:

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    # Some calculation here to alter row
    dataset.set_value(index, "FeatureName", feature_value)
```

Keep in mind that this pattern isn't very efficient. If you are going to do this, try all of your features at once.

A common *best practice* is to touch every sample as little as possible, preferably only once.

Some example features that you could try and implement are as follows:

- How many days has it been since each team's previous match? Teams may be tired if they play too many games in a short time frame.
- How many games of the last five did each team win? This will give a more stable form of the `HomeLastWin` and `VisitorLastWin` features we extracted earlier (and can be extracted in a very similar way).
- Do teams have a good record when visiting certain other teams? For instance, one team may play well in a particular stadium, even if they are the visitors.

If you are facing trouble extracting features of these types, check the pandasdocumentation at `http://pandas.pydata.org/pandas-docs/stable/` for help. Alternatively, you can try an online forum such as Stack Overflow for assistance.

More extreme examples could use player data to estimate the strength of each team's sides to predict who won. These types of complex features are used every day by gamblers and sports betting agencies to try to turn a profit by predicting the outcome of sports matches.

Summary

In this chapter, we extended our use of scikit-learn's classifiers to perform classification and introduced the `pandas`library to manage our data. We analyzed real-world data on basketball results from the NBA, saw some of the problems that even well-curated data introduces, and created new features for our analysis.

We saw the effect that good features have on performance and used an ensemble algorithm, random forests, to further improve the accuracy. To take these concepts further, try to create your own features and test them out. Which features perform better? If you have trouble coming up with features, think about what other datasets can be included. For example, if key players are injured, this might affect the results of a specific match and cause a better team to lose.

In the next chapter, we will extend the affinity analysis that we performed in the first chapter to create a program to find similar books. We will see how to use algorithms for ranking and also use an approximation to improve the scalability of data mining.

4

Recommending Movies Using Affinity Analysis

In this chapter, we will look at **affinity analysis** which determines when objects occur frequently together. This is also colloquially called market basket analysis, after one of the common use cases - determining when items are purchased together frequently in a store.

In `Chapter 3`, *Predicting Sports Winners with Decision Trees*, we looked at an object as a focus and used features to describe that object. In this chapter, the data has a different form. We have transactions where the objects of interest (movies, in this chapter) are used within those transactions in some way. The aim is to discover when objects occur simultaneously. In a case where we wish to work out when two movies are recommended by the same reviewers, we can use affinity analysis.

The key concepts of this chapter are as follows:

- Affinity analysis for product recommendations
- Feature association mining using the Apriori algorithm
- Recommendation Systems and the inherent challenges
- Sparse data formats and how to use them

Affinity analysis

Affinity analysis is the task of determining when objects are used in similar ways. In the previous chapter, we focused on whether the objects themselves are similar - in our case whether the games were similar in nature. The data for affinity analysis is often described in the form of a transaction. Intuitively, this comes from a transaction at a store—determining when objects are purchased together as a way to recommend products to users that they might purchase.

However, affinity analysis can be applied to many processes that do not use transactions in this sense:

- Fraud detection
- Customer segmentation
- Software optimization
- Product recommendations

Affinity analysis is usually much more exploratory than classification. At the very least, we often simply rank the results and choose the top five recommendations (or some other number), rather than expect the algorithm to give us a specific answer.

Furthermore, we often don't have the complete dataset we expect for many classification tasks. For instance, in movie recommendation, we have reviews from different people on different movies. However, it is highly unlikely we have each reviewer review all of the movies in our dataset. This leaves an important and difficult question in affinity analysis. If a reviewer hasn't reviewed a movie, is that an indication that they aren't interested in the movie (and therefore wouldn't recommend it) or simply that they haven't reviewed it yet?

Thinking about gaps in your datasets can lead to questions like this. In turn, that can lead to answers that may help improve the efficacy of your approach. As a budding data miner, knowing where your models and methodologies need improvement is key to creating great results.

Algorithms for affinity analysis

We introduced a basic method for affinity analysis in `Chapter 1`, *Getting Started with Data Mining*, which tested all of the possible rule combinations. We computed the confidence and support for each rule, which in turn allowed us to rank them to find the best rules.

However, this approach is not efficient. Our dataset in `Chapter 1`, *Getting Started with Data Mining*, had just five items for sale. We could expect even a small store to have hundreds of items for sale, while many online stores would have thousands (or millions!). With a naive rule creation, such as our previous algorithm from `Chapter 1`, *Getting Started with Data Mining*, the growth in the time needed to compute these rules increases exponentially. As we add more items, the time it takes to compute all rules increases significantly faster. Specifically, the total possible number of rules is $2n - 1$. For our five-item dataset, there are 31 possible rules. For 10 items, it is 1023. For just 100 items, the number has 30 digits. Even the drastic increase in computing power couldn't possibly keep up with the increases in the number of items stored online. Therefore, we need algorithms that work smarter, as opposed to computers that work harder.

The classic algorithm for affinity analysis is called the **Apriori algorithm**. It addresses the exponential problem of creating sets of items that occur frequently within a database, called **frequent itemsets**. Once these frequent itemsets are discovered, creating association rules is straightforward, which we will see later in the chapter.

The intuition behind Apriori is both simple and clever. First, we ensure that a rule has sufficient support within the dataset. Defining a minimum support level is the key parameter for Apriori. To build a frequent itemset we combine smaller frequent itemsets. For itemset **(A, B)** to have a support of at least 30, both **A** and **B** must occur at least 30 times in the database. This property extends to larger sets as well. For an itemset **(A, B, C, D)** to be considered frequent, the set **(A, B, C)** must also be frequent (as must **D**).

These frequent itemsets can be built and possible itemsets that are not frequent (of which there are many) will never be tested. This saves significant time in testing new rules, as the number of frequent itemsets is expected to be significantly fewer than the total number of possible itemsets.

Other example algorithms for affinity analysis build on this, or similar concepts, including the **Eclat** and **FP-growth** algorithms. There are many improvements to these algorithms in the data mining literature that further improve the efficiency of the method. In this chapter, we will focus on the basic Apriori algorithm.

Overall methodology

To perform association rule mining for affinity analysis, we first use the Apriori algorithm to generate frequent itemsets. Next, we create association rules (for example, *if a person recommended movie X, they would also recommend movie Y*) by testing combinations of premises and conclusions within those frequent itemsets.

1. For the first stage, the Apriori algorithm needs a value for the minimum support that an itemset needs to be considered frequent. Any itemsets with less support will not be considered.

Setting this minimum support too low will cause Apriori to test a larger number of itemsets, slowing the algorithm down. Setting it too high will result in fewer itemsets being considered frequent.

2. In the second stage, after the frequent itemsets have been discovered, association rules are tested based on their confidence. We could choose a minimum confidence level, a number of rules to return, or simply return all of them and let the user decide what to do with them.

In this chapter, we will return only rules above a given confidence level. Therefore, we need to set our minimum confidence level. Setting this too low will result in rules that have a high support, but are not very accurate. Setting this higher will result in only more accurate rules being returned, but with fewer rules being discovered overall.

Dealing with the movie recommendation problem

Product recommendation is a big business. Online stores use it to up-sell to customers by recommending other products that they could buy. Making better recommendations leads to better sales. When online shopping is selling to millions of customers every year, there is a lot of potential money to be made by selling more items to these customers.

Product recommendations, including movie and books, have been researched for many years; however, the field gained a significant boost when Netflix ran their Netflix Prize between 2007 and 2009. This competition aimed to determine if anyone can predict a user's rating of a film better than Netflix was currently doing. The prize went to a team that was just over 10 percent better than the current solution. While this may not seem like a large improvement, such an improvement would net millions to Netflix in revenue from better movie recommendations over the following years.

Obtaining the dataset

Since the inception of the Netflix Prize, Grouplens, a research group at the University of Minnesota, has released several datasets that are often used for testing algorithms in this area. They have released several versions of a movie rating dataset, which have different sizes. There is a version with 100,000 reviews, one with 1 million reviews and one with 10 million reviews.

The datasets are available from `http://grouplens.org/datasets/movielens/` and the dataset we are going to use in this chapter is the *MovieLens 100K dataset* (with 100,000 reviews). Download this dataset and unzip it in your data folder. Start a new Jupyter Notebook and type the following code:

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data", "ml-100k")
ratings_filename = os.path.join(data_folder, "u.data")
```

Ensure that `ratings_filename` points to the **u.data** file in the unzipped folder.

Loading with pandas

The `MovieLens` dataset is in a good shape; however, there are some changes from the default options in `pandas.read_csv` that we need to make. To start with, the data is separated by tabs, not commas. Next, there is no heading line. This means the first line in the file is actually data and we need to manually set the column names.

When loading the file, we set the delimiter parameter to the tab character, tell pandas not to read the first row as the header (with `header=None`) and to set the column names with given values. Let's look at the following code:

```
all_ratings = pd.read_csv(ratings_filename, delimiter="t", header=None,
names
            = ["UserID", "MovieID", "Rating", "Datetime"])
```

While we won't use it in this chapter, you can properly parse the date timestamp using the following line. Dates for reviews can be an important feature in recommendation prediction, as movies that are rated together often have more similar rankings than movies ranked separately. Accounting for this can improve models significantly.

```
all_ratings["Datetime"] = pd.to_datetime(all_ratings['Datetime'], unit='s')
```

You can view the first few records by running the following in a new cell:

```
all_ratings.head()
```

The result will come out looking something like this:

	UserID	MovieID	Rating	Datetime
0	196	242	3	1997-12-04 15:55:49
1	186	302	3	1998-04-04 19:22:22
2	22	377	1	1997-11-07 07:18:36
3	244	51	2	1997-11-27 05:02:03
4	166	346	1	1998-02-02 05:33:16

Sparse data formats

This dataset is in a sparse format. Each row can be thought of as a cell in a large feature matrix of the type used in previous chapters, where rows are users and columns are individual movies. The first column would be each user's review of the first movie, the second column would be each user's review of the second movie, and so on.

There are around 1,000 users and 1,700 movies in this dataset, which means that the full matrix would be quite large (nearly 2 million entries). We may run into issues storing the whole matrix in memory and computing on it would be troublesome. However, this matrix has the property that most cells are empty, that is, there is no review for most movies for most users. There is no review of movie number 675 for user number 213 though, and not for most other combinations of user and movie.

The format given here represents the full matrix, but in a more compact way. The first row indicates that user number 196 reviewed movie number 242, giving it a ranking of 3 (out of five) on December 4, 1997.

Any combination of user and movie that isn't in this database is assumed to not exist. This saves significant space, as opposed to storing a bunch of zeroes in memory. This type of format is called a sparse matrix format. As a rule of thumb, if you expect about 60 percent or more of your dataset to be empty or zero, a sparse format will take less space to store.

When computing on sparse matrices, the focus isn't usually on the data we don't have—comparing all of the zeroes. We usually focus on the data we have and compare those.

Understanding the Apriori algorithm and its implementation

The goal of this chapter is to produce rules of the following form: *if a person recommends this set of movies, they will also recommend this movie*. We will also discuss extensions where a person who recommends a set of movies, is likely to recommend another particular movie.

To do this, we first need to determine if a person recommends a movie. We can do this by creating a new feature **Favorable**, which is **True** if the person gave a favorable review to a movie:

```
all_ratings["Favorable"] = all_ratings["Rating"] > 3
```

We can see the new feature by viewing the dataset:

```
all_ratings[10:15]
```

	UserID	MovieID	Rating	Datetime	Favorable
10	62	257	2	1997-11-12 22:07:14	False
11	286	1014	5	1997-11-17 15:38:45	True
12	200	222	5	1997-10-05 09:05:40	True
13	210	40	3	1998-03-27 21:59:54	False
14	224	29	3	1998-02-21 23:40:57	False

We will sample our dataset to form training data. This also helps reduce the size of the dataset that will be searched, making the Apriori algorithm run faster. We obtain all reviews from the first 200 users:

```
ratings = all_ratings[all_ratings['UserID'].isin(range(200))]
```

Next, we can create a dataset of only the favorable reviews in our sample:

```
favorable_ratings_mask = ratings["Favorable"]
favorable_ratings = ratings[favorable_ratings_mask]
```

We will be searching the user's favorable reviews for our itemsets. So, the next thing we need is the movies which each user has given a favorable rating. We can compute this by grouping the dataset by the `UserID` and iterating over the movies in each group:

```
favorable_reviews_by_users = dict((k, frozenset(v.values)) for k, v in
favorable_ratings.groupby("UserID")["MovieID"])
```

In the preceding code, we stored the values as a `frozenset`, allowing us to quickly check if a movie has been rated by a user.

Sets are much faster than lists for this type of operation, and we will use them in later code.

Finally, we can create a `DataFrame` that tells us how frequently each movie has been given a favorable review:

```
num_favorable_by_movie = ratings[["MovieID",
"Favorable"]].groupby("MovieID").sum()
```

We can see the top five movies by running the following code:

```
num_favorable_by_movie.sort_values(by="Favorable", ascending=False).head()
```

Let's see the top five movies list. We only have IDs now, and will get their titles later in the chapter.

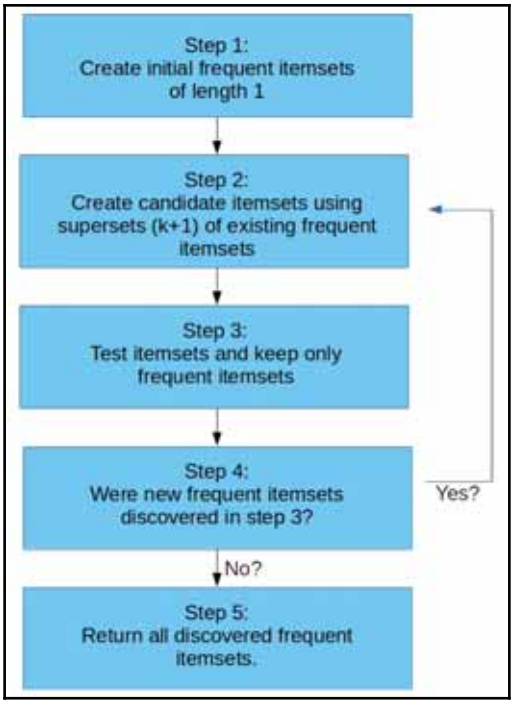
Movie ID	Favorable
50	100
100	89
258	83
181	79
174	74

Looking into the basics of the Apriori algorithm

The Apriori algorithm is part of our affinity analysis methodology and deals specifically with finding frequent itemsets within the data. The basic procedure of Apriori builds up new candidate itemsets from previously discovered frequent itemsets. These candidates are tested to see if they are frequent, and then the algorithm iterates as explained here:

1. Create initial frequent itemsets by placing each item in its own itemset. Only items with at least the minimum support are used in this step.
2. New candidate itemsets are created from the most recently discovered frequent itemsets by finding supersets of the existing frequent itemsets.
3. All candidate itemsets are tested to see if they are frequent. If a candidate is not frequent then it is discarded. If there are no new frequent itemsets from this step, go to the last step.
4. Store the newly discovered frequent itemsets and go to the second step.
5. Return all of the discovered frequent itemsets.

This process is outlined in the following workflow:



Implementing the Apriori algorithm

On the first iteration of Apriori, the newly discovered itemsets will have a length of 2, as they will be supersets of the initial itemsets created in the first step. On the second iteration (after applying the fourth step and going back to step 2), the newly discovered itemsets will have a length of 3. This allows us to quickly identify the newly discovered itemsets, as needed in the second step.

We can store our discovered frequent itemsets in a dictionary, where the key is the length of the itemsets. This allows us to quickly access the itemsets of a given length, and therefore the most recently discovered frequent itemsets, with the help of the following code:

```
frequent_itemsets = {}
```

We also need to define the minimum support needed for an itemset to be considered frequent. This value is chosen based on the dataset but try different values to see how that affects the result. I recommend only changing it by 10 percent at a time though, as the time the algorithm takes to run will be significantly different! Let's set a minimum support value:

```
min_support = 50
```

To implement the first step of the Apriori algorithm, we create an itemset with each movie individually and test if the itemset is frequent. We use `frozenset`, as they allow us to perform faster set-based operations later on, and they can also be used as keys in our counting dictionary (normal sets cannot).

Let's look at the following example of `frozenset` code:

```
frequent_itemsets[1] = dict((frozenset((movie_id,)), row["Favorable"])
 for movie_id, row in num_favorable_by_movie.iterrows()
 if row["Favorable"] > min_support)
```

We implement the second and third steps together for efficiency by creating a function that takes the newly discovered frequent itemsets, creates the supersets, and then tests if they are frequent. First, we set up the function to perform these steps:

```
from collections import defaultdict

def find_frequent_itemsets(favorable_reviews_by_users, k_1_itemsets,
min_support):
    counts = defaultdict(int)
    for user, reviews in favorable_reviews_by_users.items():
        for itemset in k_1_itemsets:
            if itemset.issubset(reviews):
                for other_reviewed_movie in reviews - itemset:
```

```
                    current_superset = itemset |
frozenset((other_reviewed_movie,))
                    counts[current_superset] += 1
    return dict([(itemset, frequency) for itemset, frequency in
counts.items() if frequency >= min_support])
```

In keeping with our rule of thumb of reading through the data as little as possible, we iterate over the dataset once per call to this function. While this doesn't matter too much in this implementation (our dataset is relatively small compared to the average computer), **single-pass** is a good practice to get into for larger applications.

Let's have a look at the core of this function in detail. We iterate through each user, and each of the previously discovered itemsets, and then check if it is a subset of the current set of reviews, which are stored in `k_1_itemsets` (note that here, **k_1** means *k-1*). If it is, this means that the user has reviewed each movie in the itemset. This is done by the `itemset.issubset(reviews)` line.

We can then go through each individual movie that the user has reviewed (that is not already in the itemset), create a superset by combining the itemset with the new movie and record that we saw this superset in our counting dictionary. These are the candidate frequent itemsets for this value of *k*.

We end our function by testing which of the candidate itemsets have enough support to be considered frequent and return only those that have a support more than our `min_support` value.

This function forms the heart of our Apriori implementation and we now create a loop that iterates over the steps of the larger algorithm, storing the new itemsets as we increase *k* from 1 to a maximum value. In this loop, **k** represents the length of the soon-to-be discovered frequent itemsets, allowing us to access the previously most discovered ones by looking in our frequent_itemsets dictionary using the key *k - 1*. We create the frequent itemsets and store them in our dictionary by their length. Let's look at the code:

```
for k in range(2, 20):
    # Generate candidates of length k, using the frequent itemsets of
length k-1
    # Only store the frequent itemsets
    cur_frequent_itemsets =
find_frequent_itemsets(favorable_reviews_by_users,
                                                    frequent_itemsets[k-1],
min_support)
    if len(cur_frequent_itemsets) == 0:
        print("Did not find any frequent itemsets of length {}".format(k))
        sys.stdout.flush()
        break
```

```
    else:
        print("I found {} frequent itemsets of length
{}".format(len(cur_frequent_itemsets), k))
        sys.stdout.flush()
        frequent_itemsets[k] = cur_frequent_itemsets
```

If we do find frequent itemsets, we print out a message to let us know the loop will be running again. If we don't, we stop iterating, as there cannot be frequent itemsets for *k+1* if there are no frequent itemsets for the current value of *k*, therefore we finish the algorithm.

We use `sys.stdout.flush()` to ensure that the printouts happen while the code is still running. Sometimes, in large loops in particular cells, the printouts will not happen until the code has completed. Flushing the output in this way ensures that the printout happens when we want, rather than when the interface decides it can allocate the time to print. Don't flush too frequently though—the flush operation carries a computational cost (as does normal printing) and this will slow down the program.

You can now run the above code.
The preceding code returns about 2000 frequent itemsets of varying lengths. You'll notice that the number of itemsets grows as the length increases before it shrinks. It grows because of the increasing number of possible rules. After a while, the large number of combinations no longer has the support necessary to be considered frequent. This results in the number shrinking. This shrinking is the benefit of the Apriori algorithm. If we search all possible itemsets (not just the supersets of frequent ones), we would be searching thousands of times more itemsets to see if they are frequent.

Even if this shrinking didn't occur, the algorithm meets an absolute end when rules for a combination of all movies together is discovered. Therefore the Apriori algorithm will always terminate.

It may take a few minutes for this code to run, more if you have older hardware. If you find you are having trouble running any of the code samples, take a look at using an online cloud provider for additional speed. Details about using the cloud to do the work are given in Appendix, Next Steps.

Extracting association rules

After the Apriori algorithm has completed, we have a list of frequent itemsets. These aren't exactly association rules, but they can easily be converted into these rules. A frequent itemset is a set of items with a minimum support, while an association rule has a premise and a conclusion. The data is the same for the two.

We can make an *association rule from a frequent itemset* by taking one of the movies in the itemset and denoting it as the conclusion. The other movies in the itemset will be the premise. This will form rules of the following form: *if a reviewer recommends all of the movies in the premise, they will also recommend the conclusion movie*.

For each itemset, we can generate a number of association rules by setting each movie to be the conclusion and the remaining movies as the premise.

In code, we first generate a list of all of the rules from each of the frequent itemsets, by iterating over each of the discovered frequent itemsets of each length. We then iterate over every movie in the itemset, as the conclusion.

```
candidate_rules = []
for itemset_length, itemset_counts in frequent_itemsets.items():
    for itemset in itemset_counts.keys():
        for conclusion in itemset:
            premise = itemset - set((conclusion,))
            candidate_rules.append((premise, conclusion))
```

This returns a very large number of candidate rules. We can see some by printing out the first few rules in the list:

```
print(candidate_rules[:5])
```

The resulting output shows the rules that were obtained:

```
[(frozenset({79}), 258), (frozenset({258}), 79), (frozenset({50}), 64),
(frozenset({64}), 50), (frozenset({127}), 181)]
```

In these rules, the first part (the `frozenset`) is the list of movies in the premise, while the number after it is the conclusion. In the first case, if a reviewer recommends movie 79, they are also likely to recommend movie 258.

Next, we compute the confidence of each of these rules. This is performed much like in `Chapter 1`, *Getting Started with Data Mining*, with the only changes being those necessary for computing using the new data format.

The process of computing confidence starts by creating dictionaries to store how many times we see the premise leading to the conclusion (a correct example of the rule) and how many times it doesn't (an incorrect example). We then iterate over all reviews and rules, working out whether the premise of the rule applies and, if it does, whether the conclusion is accurate.

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
for user, reviews in favorable_reviews_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

We then compute the confidence for each rule by dividing the correct count by the total number of times the rule was seen:

```
rule_confidence = {candidate_rule:
                   (correct_counts[candidate_rule] /
float(correct_counts[candidate_rule] +
                     incorrect_counts[candidate_rule]))
                  for candidate_rule in candidate_rules}
```

Now we can print the top five rules by sorting this confidence dictionary and printing the results:

```
from operator import itemgetter
sorted_confidence = sorted(rule_confidence.items(), key=itemgetter(1),
reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print("Rule: If a person recommends {0} they will also recommend
{1}".format(premise, conclusion))
    print(" - Confidence: {0:.3f}".format(rule_confidence[(premise,
conclusion)]))
    print("")
```

The resulting printout shows only the movie IDs, which isn't very helpful without the names of the movies also. The dataset came with a file called **u.items**, which stores the movie names and their corresponding **MovieID** (as well as other information, such as the genre).

We can load the titles from this file using pandas. Additional information about the file and categories is available in the **README** file that came with the dataset. The data in the files is in CSV format, but with data separated by the **|** symbol; it has no header and the encoding is important to set. The column names were found in the **README** file.

```
movie_name_filename = os.path.join(data_folder, "u.item")
movie_name_data = pd.read_csv(movie_name_filename, delimiter="|",
header=None,
                              encoding = "mac-roman")
movie_name_data.columns = ["MovieID", "Title", "Release Date", "Video
Release", "IMDB", "<UNK>",
                           "Action", "Adventure", "Animation",
"Children's", "Comedy", "Crime",
                           "Documentary", "Drama", "Fantasy", "Film-Noir",
"Horror", "Musical",
                           "Mystery", "Romance", "Sci-Fi", "Thriller",
"War", "Western"]
```

Getting the movie title is an important and frequently used step, therefore it makes sense to turn it into a function. We will create a function that will return a movie's title from its **MovieID**, saving us the trouble of looking it up each time. Let's look at the code:

```
def get_movie_name(movie_id):
    title_object = movie_name_data[movie_name_data["MovieID"] ==
movie_id]["Title"]
    title = title_object.values[0]
    return title
```

In a new Jupyter Notebook cell, we adjust our previous code for printing out the top rules to also include the titles:

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx in premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will also recommend
{1}".format(premise_names, conclusion_name))
    print(" - Confidence: {0:.3f}".format(rule_confidence[(premise,
conclusion)]))
    print("")
```

The result is much more readable (there are still some issues, but we can ignore them for now):

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Silence of
the Lambs, The (1991), Pulp Fiction (1994), Star Wars (1977), Twelve
Monkeys (1995) they will also recommend Raiders of the Lost Ark (1981)
 - Confidence: 1.000

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star Wars
(1977), Pulp Fiction (1994) they will also recommend Twelve Monkeys (1995)
 - Confidence: 1.000

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the Lost
Ark (1981), Twelve Monkeys (1995) they will also recommend Star Wars (1977)
 - Confidence: 1.000

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence of
the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire Strikes
Back, The (1980), Star Wars (1977) they will also recommend Raiders of the
Lost Ark (1981)
 - Confidence: 1.000

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy Story
(1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980), Fugitive,
The (1993), Star Wars (1977) they will also recommend Return of the Jedi
(1983)
 - Confidence: 1.000
```

Evaluating the association rules

In a broad sense, we can evaluate the association rules using the same concept as for classification. We use a test set of data that was not used for training, and evaluate our discovered rules based on their performance in this test set.

To do this, we will compute the test set confidence, that is, the confidence of each rule on the testing set. We won't apply a formal evaluation metric in this case; we simply examine the rules and look for good examples.

Formal evaluation could include a classification accuracy by determining the accuracy of predicting whether a user rates a given movie as favorable. In this case, as described below, we will informally look at the rules to find those that are more reliable:

1. First, we extract the test dataset, which is all of the records that we didn't use in the training set. We used the first 200 users (by ID value) for the training set, and we will use all of the rest for the testing dataset. As with the training set, we will also get the favorable reviews for each of the users in this dataset as well. Let's look at the code:

```
test_dataset = all_ratings[~all_ratings['UserID'].isin(range(200))]
test_favorable = test_dataset[test_dataset["Favorable"]]
test_favorable_by_users = dict((k, frozenset(v.values)) for k, v in
                               test_favorable.groupby("UserID")["MovieID"])
```

2. We then count the correct instances where the premise leads to the conclusion, in the same way that we did before. The only change here is the use of the test data instead of the training data. Let's look at the code:

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
for user, reviews in test_favorable_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

3. Next, we compute the confidence of each rule from the correct counts and sort them. Let's look at the code:

```
test_confidence = {candidate_rule:
                               (correct_counts[candidate_rule] /
float(correct_counts[candidate_rule] + incorrect_counts[candidate_rule]))
                               for candidate_rule in rule_confidence}
sorted_test_confidence = sorted(test_confidence.items(), key=itemgetter(1),
reverse=True)
```

4. Finally, we print out the best association rules with the titles instead of the movie IDs:

```
for index in range(10):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
```

```
    premise_names = ", ".join(get_movie_name(idx) for idx in premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will also recommend
{1}".format(premise_names, conclusion_name))
    print(" - Train Confidence:
{0:.3f}".format(rule_confidence.get((premise, conclusion), -1)))
    print(" - Test Confidence:
{0:.3f}".format(test_confidence.get((premise, conclusion), -1)))
    print("")
```

We can now see which rules are most applicable in new unseen data:

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Silence of
the Lambs, The (1991), Pulp Fiction (1994), Star Wars (1977), Twelve
Monkeys (1995) they will also recommend Raiders of the Lost Ark (1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.909

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star Wars
(1977), Pulp Fiction (1994) they will also recommend Twelve Monkeys (1995)
 - Train Confidence: 1.000
 - Test Confidence: 0.609

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the Lost
Ark (1981), Twelve Monkeys (1995) they will also recommend Star Wars (1977)
 - Train Confidence: 1.000
 - Test Confidence: 0.946

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence of
the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire Strikes
Back, The (1980), Star Wars (1977) they will also recommend Raiders of the
Lost Ark (1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.971

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy Story
(1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980), Fugitive,
The (1993), Star Wars (1977) they will also recommend Return of the Jedi
(1983)
 - Train Confidence: 1.000
 - Test Confidence: 0.900
```

The second rule, for instance, has a perfect confidence in the training data, but it is only accurate in 60 percent of cases for the test data. Many of the other rules in the top 10 have high confidences in test data, making them good rules for making recommendations.

You may also notice that these movies tend to be very popular and good films. This gives us a baseline algorithm that we could compare against, i.e. instead of trying to do personalized recommendations, just recommend the most liked movies overall. Have a shot at implementing this algorithm - does the Apriori algorithm outperform it and by how much? Another baseline could be to simply recommend movies at random from the same genre.

If you are looking through the rest of the rules, some will have a test confidence of -1. Confidence values are always between 0 and 1. This value indicates that the particular rule wasn't found in the test dataset at all.

Summary

In this chapter we performed affinity analysis in order to recommend movies based on a large set of reviewers. We did this in two stages. First, we found frequent itemsets in the data using the Apriori algorithm. Then, we created association rules from those itemsets.

The use of the Apriori algorithm was necessary due to the size of the dataset. In `Chapter 1`, *Getting Started With Data Mining*, we used a brute-force approach, which has exponential growth in the time needed to compute those rules required for a smarter approach. This is a common pattern for data mining: we can solve many problems in a brute force manner for small datasets, but smarter algorithms are required to apply the concepts to larger datasets.

We performed training on a subset of our data in order to find the association rules, and then tested those rules on the rest of the data—a testing set. From what we discussed in the previous chapters, we could extend this concept to use cross-fold validation to better evaluate the rules. This would lead to a more robust evaluation of the quality of each rule.

To take the concepts in this chapter further, investigate which movies obtain high overall scores (i.e. lots of recommendations), but do not have adequate rules to recommend them to new users. How would you alter the algorithm to recommend these movies?

So far, all of our datasets have been described in terms of features. However, not all datasets are *pre-defined* in this way. In the next chapter, we will look at scikit-learn's transformers (they were introduced in *Chapter 3, Predicting Sports Winners with Decision Trees*) as a way to extract features from data. We will discuss how to implement our own transformers, extend existing ones, and concepts we can implement using them.

5
Features and scikit-learn Transformers

The datasets we have used so far have been described in terms of *features*. In the previous chapter, we used a transaction-centric dataset. However, ultimately this was just a different format for representing feature-based data.

There are many other types of datasets, including text, images, sounds, movies, or even real objects. Most data mining algorithms rely on having numerical or categorical features. This means we need a way to represent these types before we input them into the data mining algorithm. We call this representation a **model**.

In this chapter, we will discuss how to extract numerical and categorical features, and choose the best features when we do have them. We will discuss some common patterns and techniques for extracting features. Choosing your model appropriately is critically important to the outcome of the data mining exercise, more so than the choice of classification algorithm.

The key concepts introduced in this chapter include:

- Extracting features from datasets
- Creating models for your data
- Creating new features
- Selecting good features
- Creating your own transformer for custom datasets

Feature extraction

Extracting features is one of the most critical tasks in data mining, and it generally affects your end result more than the choice of data mining algorithm. Unfortunately, there are no hard and fast rules for choosing features that will result in high-performance data mining. The choice of features determines the model that you are using to represent your data.

Model creation is where the science of data mining becomes more of an art and why automated methods of performing data mining (there are several methods of this type) focus on algorithm choice and not model creation. Creating good models relies on intuition, domain expertise, data mining experience, trial and error, and sometimes a little luck.

Representing reality in models

Given what we have done so far in the book, it is easy to forget that the reason we are performing data mining is to affect real world objects, not just manipulating a matrix of values. Not all datasets are presented in terms of features. Sometimes, a dataset consists of nothing more than all of the books that have been written by a given author. Sometimes, it is the film of each of the movies released in 1979. At other times, it is a library collection of interesting historical artifacts.

From these datasets, we may want to perform a data mining task. For the books, we may want to know the different categories that the author writes. In the films, we may wish to see how women are portrayed. In the historical artifacts, we may want to know whether they are from one country or another. It isn't possible to just pass these raw datasets into a decision tree and see what the result is.

For a data mining algorithm to assist us here, we need to represent these as **features**. Features are a way to create a model and the model provides an approximation of reality in a way that data mining algorithms can understand. Therefore, a model is just a simplified version of some aspect of the real world. As an example, the game of chess is a simplified model (in game form) for historical warfare.

Selecting features has another advantage: they reduce the complexity of the real world into a more manageable model.

Imagine how much information it would take to properly, accurately, and fully describe a real-world object to someone that has no background knowledge of the item. You would need to describe the size, weight, texture, composition, age, flaws, purpose, origin, and so on.

As the complexity of real objects is too much for current algorithms, we use these simpler models instead.

This simplification also focuses our intent in the data mining application. In later chapters, we will look at **clustering** and where it is critically important. If you put random features in, you will get random results out.

However, there is a downside as this simplification reduces the detail, or may remove good indicators of the things we wish to perform data mining on.

Thought should always be given to how to represent reality in the form of a model. Rather than just using what has been used in the past, you need to consider the goal of the data mining exercise. What are you trying to achieve? In `Chapter 3`, *Predicting Sports Winners with Decision Trees*, we created features by thinking about the goal (predicting winners) and used a little domain knowledge to come up with ideas for new features.

Not all features need to be numeric or categorical. Algorithms have been developed that work directly on text, graphs, and other data structures. Unfortunately, those algorithms are outside the scope of this book. In this book, and normally in your data mining career, we mainly use numeric or categorical features.

The *Adult* dataset is a great example of taking a complex reality and attempting to model it using features. In this dataset, the aim is to estimate if someone earns more than $50,000 per year.

To download the dataset, navigate to `http://archive.ics.uci.edu/ml/datasets/Adult` and click on the **Data Folder** link. Download the `adult.data` and `adult.names` into a directory named Adult in your data folder.

This dataset takes a complex task and describes it in features. These features describe the person, their environment, their background, and their life status.

Open a new Jupyter Notebook for this chapter, set the data filename and load the data with pandas:

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data", "Adult")
adult_filename = os.path.join(data_folder, "adult.data")

adult = pd.read_csv(adult_filename, header=None, names=["Age", "Work-
Class", "fnlwgt",
                    "Education", "Education-Num", "Marital-Status",
"Occupation",
                    "Relationship", "Race", "Sex", "Capital-gain",
"Capital-loss",
                    "Hours-per-week", "Native-Country", "Earnings-Raw"])
```

Most of the code is the same as in the previous chapters.

Don't want to type those heading names? Don't forget you can download the code from Packt Publishing, or alternatively from the author's GitHub repository for this book: `https://github.com/dataPipelineAU/LearningDataMiningWithPython2`

The adult file itself contains two blank lines at the end of the file. By default, pandas will interpret the penultimate new line to be an empty (but valid) row. To remove this, we remove any line with invalid numbers (the use of `inplace` just makes sure the same Dataframe is affected, rather than creating a new one):

```
adult.dropna(how='all', inplace=True)
```

Having a look at the dataset, we can see a variety of features from `adult.columns`:

```
adult.columns
```

The results show each of the feature names that are stored inside an Index object from pandas:

```
Index(['Age', 'Work-Class', 'fnlwgt', 'Education',
'Education-Num', 'Marital-Status', 'Occupation', 'Relationship',
'Race', 'Sex', 'Capital-gain', 'Capital-loss', 'Hours-per-week',
'Native-Country', 'Earnings-Raw'], dtype='object')
```

Common feature patterns

While there are millions of ways to create models, there are some common patterns that are employed across different disciplines. However, choosing appropriate features is tricky and it is worth considering how a feature might correlate to the end result. As a well known adage goes, *don't judge a book by its cover*—it is probably not worth considering the size of a book if you are interested in the message contained within.

Some commonly used features focus on the physical properties of the real world objects being studied, for example:

- Spatial properties such as the length, width, and height of an object
- Weight and/or density of the object
- Age of an object or its components
- The type of the object
- The quality of the object

Other features might rely on the usage or history of the object:

- The producer, publisher, or creator of the object
- The year of manufacturing

Other features describe a dataset in terms of its components:

- Frequency of a given subcomponent, such as a word in a book
- Number of subcomponents and/or the number of different subcomponents
- Average size of the subcomponents, such as the average sentence length

Ordinal features allow us to perform ranking, sorting, and grouping of similar values. As we have seen in previous chapters, **features** can be numerical or categorical.

Numerical features are often described as being ordinal. For example, three people, Alice, Bob, and Charlie, may have heights of 1.5 m, 1.6 m, and 1.7 m. We would say that Alice and Bob are more similar in height than Alice and Charlie.

The Adult dataset that we loaded in the last section contains examples of continuous, ordinal features. For example, the **Hours-per-week** feature tracks how many hours per week people work. Certain operations make sense on a feature like this. They include computing the mean, standard deviation, minimum, and maximum. There is a function in pandas for giving some basic summary stats of this type:

```
adult["Hours-per-week"].describe()
```

The result tells us a little about this feature:

```
count 32561.000000
mean 40.437456
std 12.347429
min 1.000000
25% 40.000000
50% 40.000000
75% 45.000000
max 99.000000
dtype: float64
```

Some of these operations do not make sense for other features. For example, it doesn't make sense to compute the sum of the education statuses of these people. In contrast, it would make sense to compute the sum of the number of orders by each customer on an online store.

There are also features that are not numerical, but still ordinal. The **Education** feature in the Adult dataset is an example of this. For example, a Bachelor's degree is a higher education status than finishing high school, which is a higher status than not completing high school. It doesn't quite make sense to compute the mean of these values, but we can create an approximation by taking the median value. The dataset gives a helpful feature, `Education-Num`, which assigns a number that is basically equivalent to the number of years of education completed. This allows us to quickly compute the median:

```
adult["Education-Num"].median()
```

The result is 10, or finishing one year past high school. If we didn't have this, we could compute the median by creating an ordering over the education values.

Features can also be categorical. For instance, a ball can be a tennis ball, cricket ball, football, or any other type of ball. Categorical features are also referred to as nominal features. For nominal features, the values are either the same or they are different. While we could rank balls by size or weight, just the category alone isn't enough to compare things. A tennis ball is not a cricket ball, and it is also not a football. We could argue that a tennis ball is more similar to a cricket ball (say, in size), but the category alone doesn't differentiate this—they are the same, or they are not.

We can convert categorical features to numerical features using the one-hot encoding, as we saw in `Chapter 3`, *Predicting Sports Winners with Decision Trees*. For the aforementioned categories of balls, we can create three new binary features: is a tennis ball, is a cricket ball, and is a football. This process is the one-hot encoding we used in `Chapter 3`, *Predicting Sports Winners with Decision Trees*. For a tennis ball, the vector would be `[1, 0, 0]`. A cricket ball has the values `[0, 1, 0]`, while a football has the value `[0, 0, 1]`. These are binary features but can be used as continuous features by many algorithms. One key reason for doing this is that it easily allows for direct numerical comparison (such as computing the distance between samples).

The Adult dataset contains several categorical features, with **Work-Class** being one example. While we could argue that some values are of higher rank than others (for instance, a person with a job is likely to have a better income than a person without), it doesn't make sense for all values. For example, a person working for the state government is not more or less likely to have a higher income than someone working in the private sector.

We can view the unique values for this feature in the dataset using the `unique()` function:

```
adult["Work-Class"].unique()
```

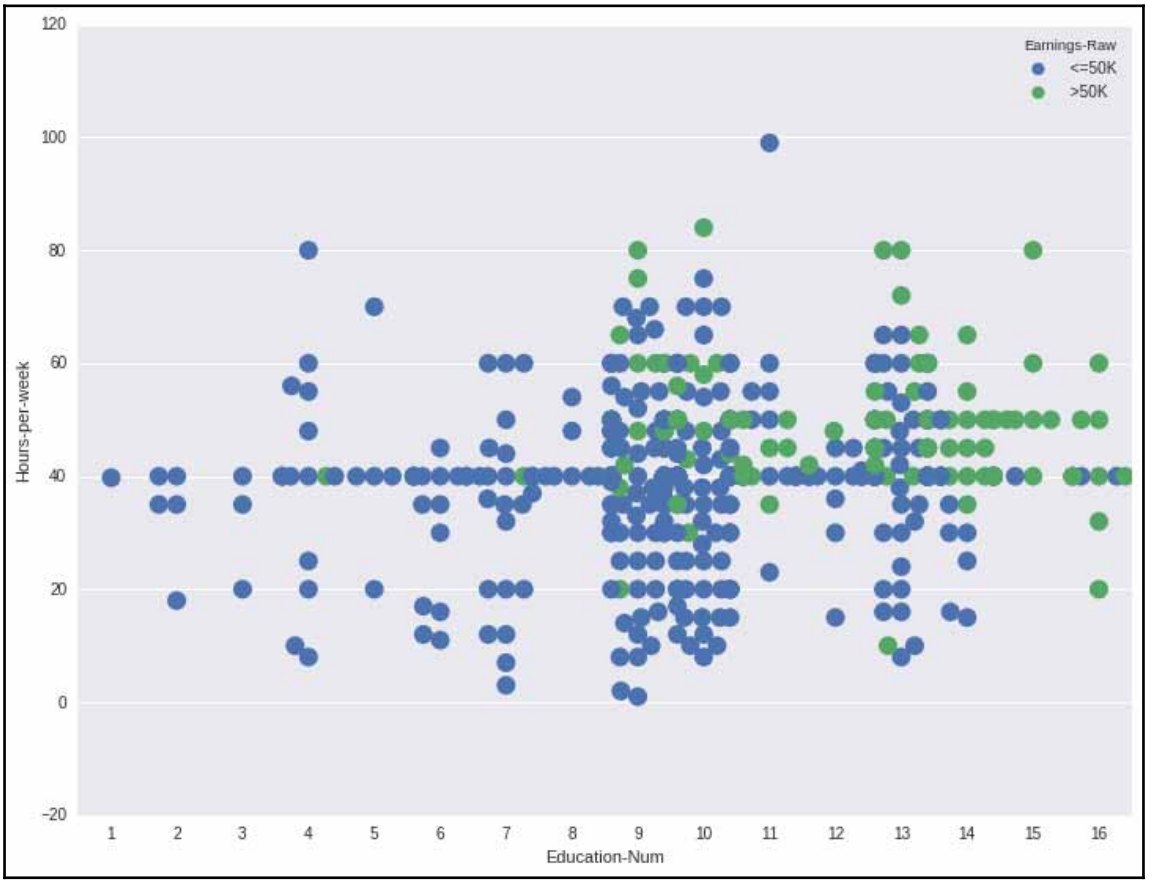
The result shows the unique values in this column:

```
array([' State-gov', ' Self-emp-not-inc', ' Private', ' Federal-gov',
' Local-gov', ' ?', ' Self-emp-inc', ' Without-pay',
' Never-worked', nan], dtype=object)
```

There are some missing values in the preceding data, but they won't affect our computations in this example. You can also use the `adult.value_counts()` function to see how frequently each value appears.

Another really useful step to take with a new dataset is to visualise it. The following code will create a swarm plot, giving a view of how education and hours-worked relate to the final classification (identified by colour):

```
%matplotlib inline
import seaborn as sns
from matplotlib import pyplot as plt
sns.swarmplot(x="Education-Num", y="Hours-per-week", hue="Earnings-Raw",
data=adult[::50])
```



In the above code, we sample the dataset to show every 50 rows, using the `adult[::50]` dataset indexing. Setting this to just `adult` will result in all samples being shown, but that may also make the graph hard to read.

Similarly, we can convert numerical features to categorical features through a process called **discretization**, as we saw in `Chapter 1`, *Getting Started With Data Mining*. We can call any person who is taller than 1.7 m tall, and any person shorter than 1.7 m short. This gives us a categorical feature (although still an ordinal one). We do lose some data here. For instance, two people, one 1.69 m tall and one 1.71 m, will be in two different categories and considered drastically different from each other by our algorithm. In contrast, a person 1.2 m tall will be considered of roughly the same height as the person 1.69 m tall! This loss of detail is a side effect of discretization, and it is an issue that we deal with when creating models.

In the Adult dataset, we can create a `LongHours` feature, which tells us if a person works more than 40 hours per week. This turns our continuous feature (`Hours-per-week`) into a categorical one that is True if the number of hours is more than 40, False otherwise:

```
adult["LongHours"] = adult["Hours-per-week"] > 40
```

Creating good features

Simplification due to modeling is a key reason we do not have data mining methods that can just simply be applied to any dataset. A good data mining practitioner will need, or obtain, domain knowledge in the area they are applying data mining. They will look at the problem, the available data, and come up with a model that represents what they are trying to achieve.

For instance, a person's height feature may describe one component of a person, such as their ability to play basketball, but may not describe their academic performance well. If we were attempting to predict a person's grade, we may not bother measuring each person's height.

This is where data mining becomes more art than science. Extracting good features is difficult and is the topic of significant and ongoing research. Choosing better classification algorithms can improve the performance of a data mining application, but choosing better features is often a better option.

In all data mining applications, you should first outline what you are looking for before you start designing the methodology that will find it. This will dictate the types of features you are aiming for, the types of algorithms that you can use, and the expectations in the final result.

Feature selection

After initial modeling, we will often have a large number of features to choose from, but we wish to select only a small subset. There are many possible reasons for this:

- **Reducing complexity**: Many data mining algorithms need significantly more time and resources when the number of features increase. Reducing the number of features is a great way to make an algorithm run faster or with fewer resources.
- **Reducing noise**: Adding extra features doesn't always lead to better performance. Extra features may confuse the algorithm, finding correlations and patterns in training data that do not have any actual meaning. This is common in both smaller and larger datasets. Choosing only appropriate features is a good way to reduce the chance of random correlations that have no real meaning.
- **Creating readable models**: While many data mining algorithms will happily compute an answer for models with thousands of features, the results may be difficult to interpret for a human. In these cases, it may be worth using fewer features and creating a model that a human can understand.

Some classification algorithms can handle data with issues such as those described before. Getting the data right and getting the features to effectively describe the dataset you are modeling can still assist algorithms.

There are some basic tests we can perform, such as ensuring that the features are at least different. If a feature's values are all the same, it can't give us extra information to perform our data mining.

The **VarianceThreshold** transformer in `scikit-learn`, for instance, will remove any feature that doesn't have at least a minimum level of variance in the values. To show how this works, we first create a simple matrix using **NumPy**:

```
import numpy as np
X = np.arange(30).reshape((10, 3))
```

The result is the numbers 0 to 29, in three columns and 10 rows. This represents a synthetic dataset with 10 samples and three features:

```
array([[ 0, 1, 2],
[ 3, 4, 5],
[ 6, 7, 8],
[ 9, 10, 11],
[12, 13, 14],
[15, 16, 17],
[18, 19, 20],
```

```
[21, 22, 23],
[24, 25, 26],
[27, 28, 29]])
```

Then, we set the entire second column/feature to the value 1:

```
X[:,1] = 1
```

The result has lots of variance in the first and third rows, but no variance in the second row:

```
array([[ 0, 1, 2],
[ 3, 1, 5],
[ 6, 1, 8],
[ 9, 1, 11],
[12, 1, 14],
[15, 1, 17],
[18, 1, 20],
[21, 1, 23],
[24, 1, 26],
[27, 1, 29]])
```

We can now create a `VarianceThreshold` transformer and apply it to our dataset:

```
from sklearn.feature_selection import VarianceThreshold
vt = VarianceThreshold()
Xt = vt.fit_transform(X)
```

Now, the result `Xt` does not have the second column:

```
array([[ 0, 2],
[ 3, 5],
[ 6, 8],
[ 9, 11],
[12, 14],
[15, 17],
[18, 20],
[21, 23],
[24, 26],
[27, 29]])
```

We can observe the variances for each column by printing the `vt.variances_` attribute:

```
print(vt.variances_)
```

The result shows that while the first and third column contains at least some information, the second column had no variance:

```
array([ 74.25,  0. ,  74.25])
```

A simple and obvious test like this is always good to run when seeing data for the first time. Features with no variance do not add any value to a data mining application; however, they can slow down the performance of the algorithm and reduce the efficacy.

Selecting the best individual features

If we have a number of features, the problem of finding the best subset is a difficult task. It relates to solving the data mining problem itself, multiple times. As we saw in `Chapter 4`, *Recommending Movies Using Affinity Analysis*, subset-based tasks increase exponentially as the number of features increase. This exponential growth in the time needed is also true for finding the best subset of features.

One basic workaround to this problem is not to look for a subset that works well together, rather than just finding the best individual features. This univariate feature selection gives us a score based on how well a feature performs by itself. This is usually done for classification tasks, and we generally measure some type of association between a variable and the target class.

The scikit-learn package has a number of transformers for performing univariate feature selection. They include **SelectKBest**, which returns the k-best-performing features, and **SelectPercentile**, which returns the top **R%** of features. In both cases, there are a number of methods of computing the quality of a feature.

There are many different methods to compute how effectively a single feature correlates with a class value. A commonly used method is the **chi-squared ($\chi 2$) test.** Other methods include mutual information and entropy.

We can observe single-feature tests in action using our Adult dataset. First, we extract a dataset and class values from our **pandas** DataFrame. We get a selection of the features:

```
X = adult[["Age", "Education-Num", "Capital-gain", "Capital-loss", "Hours-
per-week"]].values
```

We will also create a target class array by testing whether the **Earnings-Raw** value is above $50,000 or not. If it is, the class will be True. Otherwise, it will be False. Let's look at the code:

```
y = (adult["Earnings-Raw"] == ' >50K').values
```

Next, we create our transformer using the chi2 function and a **SelectKBest** transformer:

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
transformer = SelectKBest(score_func=chi2, k=3)
```

Running `fit_transform` will call fit and then transform with the same dataset. The result will create a new dataset, choosing only the best three features. Let's look at the code:

```
Xt_chi2 = transformer.fit_transform(X, y)
```

The resulting matrix now only contains three features. We can also get the scores for each column, allowing us to find out which features were used. Let's look at the code:

```
print(transformer.scores_)
```

The printed results give us these scores:

```
[ 8.60061182e+03 2.40142178e+03 8.21924671e+07
1.37214589e+066.47640900e+03]
```

The highest values are for the first, third, and fourth columns Correlates to the Age, Capital-Gain, and Capital-Loss features. Based on a univariate feature selection, these are the best features to choose.

If you'd like to find out more about the features in the Adult dataset, take a look at the **adult.names** file that comes with the dataset and the academic paper it references.

We could also implement other correlations, such as the Pearson's correlation coefficient. This is implemented in **SciPy**, a library used for scientific computing (**scikit-learn** uses it as a base).

If scikit-learn is working on your computer, so is SciPy. You do not need to install anything further to get this sample working.

First, we import the `pearsonr` function from **SciPy**:

```
from scipy.stats import pearsonr
```

The preceding function almost fits the interface needed to be used in scikit-learn's univariate transformers. The function needs to accept two arrays (x and y in our example) as parameters and returns two arrays, the scores for each feature and the corresponding p-values. The chi2 function we used earlier only uses the required interface, which allowed us to just pass it directly to **SelectKBest.**

The **pearsonr** function in SciPy accepts two arrays; however, the X array it accepts is only one dimension. We will write a wrapper function that allows us to use this for multivariate arrays like the one we have. Let's look at the code:

```
def multivariate_pearsonr(X, y):
    scores, pvalues = [], []
    for column in range(X.shape[1]):
        # Compute the Pearson correlation for this column only
        cur_score, cur_p = pearsonr(X[:,column], y)
        # Record both the score and p-value.
        scores.append(abs(cur_score))
        pvalues.append(cur_p)
    return (np.array(scores), np.array(pvalues))
```

The Pearson value could be between -1 and 1. A value of 1 implies a perfect correlation between two variables, while a value of -1 implies a perfect negative correlation, that is, high values in one variable give low values in the other and vice versa. Such features are really useful to have. For this reason, we have stored the absolute value in the scores array, rather than the original, signed value.

Now, we can use the transformer class as before to rank the features using the Pearson correlation coefficient:

```
transformer = SelectKBest(score_func=multivariate_pearsonr, k=3)
Xt_pearson = transformer.fit_transform(X, y)
print(transformer.scores_)
```

This returns a different set of features! The features chosen this way are the first, second, and fifth columns: the **Age**, **Education**, and **Hours-per-week** worked. This shows that there is not a definitive answer to what the best features are— it depends on the metric used and the process undertaken.

We can see which feature set is better by running them through a classifier. Keep in mind that the results only indicate which subset is better for a particular classifier and/or feature combination—there is rarely a case in data mining where one method is strictly better than another in all cases! Let's look at the code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import cross_val_score
clf = DecisionTreeClassifier(random_state=14)
scores_chi2 = cross_val_score(clf, Xt_chi2, y, scoring='accuracy')
scores_pearson = cross_val_score(clf, Xt_pearson, y, scoring='accuracy')

print("Chi2 score: {:.3f}".format(scores_chi2.mean()))
print("Pearson score: {:.3f}".format(scores_pearson.mean()))
```

The chi2 average here is 0.83, while the Pearson score is lower at 0.77. For this combination, chi2 returns better results!

It is worth remembering the goal of this particular data mining activity: predicting wealth. Using a combination of good features and feature selection, we can achieve 83 percent accuracy using just three features of a person!

Feature creation

Sometimes, just selecting features from what we have isn't enough. We can create features in different ways from features we already have. The one-hot encoding method we saw previously is an example of this. Instead of having category features with options A, B, and C, we would create three new features *Is it A?*, *Is it B?*, *Is it C?*.

Creating new features may seem unnecessary and to have no clear benefit—after all, the information is already in the dataset and we just need to use it. However, some algorithms struggle when features correlate significantly, or if there are redundant features. They may also struggle if there are redundant features. For this reason, there are various ways to create new features from the features we already have.

We are going to load a new dataset, so now is a good time to start a new Jupyter Notebook. Download the **Advertisements** dataset from `http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements` and save it to your Data folder.

Next, we need to load the dataset with **pandas**. First, we set the data's filename as always:

```
import os
import numpy as np
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data")
data_filename = os.path.join(data_folder, "Ads", "ad.data")
```

There are a couple of issues with this dataset that stop us from loading it easily. You can see these issues by trying to load the dataset with `pd.read_csv`. First, the first few features are numerical, but **pandas** will load them as strings. To fix this, we need to write a converting function that will convert strings to numbers if possible. Otherwise, we will get a **Not a Number** (**NaN)** - an invalid value, which is a special value that indicates that the value could not be interpreted as a number. It is similar to none or null in other programming languages.

Another issue with this dataset is that some values are missing. These are represented in the dataset using the string **?**. Luckily, the question mark doesn't convert to a float, so we can convert those to **NaNs** using the same concept. In further chapters, we will look at other ways of dealing with missing values like this.

We will create a function that will do this conversion for us. It attempts to convert the number to a float, and if that fails, it returns NumPy's special NaN value that can be stored in place of a float:

```
def convert_number(x):
    try:
        return float(x)
    except ValueError:
        return np.nan
```

Now, we create a dictionary for the conversion. We want to convert all of the features to floats:

```
converters = {}
for i in range(1558):
    converters[i] = convert_number
```

Also, we want to set the final column, the class, (column index #1558) to a binary feature. In the Adult dataset, we created a new feature for this. In the dataset, we will convert the feature while we load it:

```
converters[1558] = lambda x: 1 if x.strip() == "ad." else 0
```

Now we can load the dataset using `read_csv`. We use the converters parameter to pass our custom conversion into **pandas**:

```
ads = pd.read_csv(data_filename, header=None, converters=converters)
```

The resulting dataset is quite large, with 1,559 features and more than 3,000 rows. Here are some of the feature values, the first five, printed by inserting `ads.head()` into a new cell:

```
ads.head()
```

	0	1	2	3	4	5	6	7	8	9	...	1549	1550	1551	1552	1553	1554	1555	1556	1557	1558
0	125.0	125.0	1.0000	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
1	57.0	468.0	8.2105	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	33.0	230.0	6.9696	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
3	60.0	468.0	7.8000	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
4	60.0	468.0	7.8000	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1

5 rows × 1559 columns

This dataset describes images on websites, with the goal of determining whether a given image is an advertisement or not.

The features in this dataset are not described well by their headings. There are two files accompanying the **ad.data** file that have more information: `ad.DOCUMENTATION` and `ad.names`. The first three features are the height, width, and ratio of the image size. The final feature is 1 if it is an advertisement and 0 if it is not.

The other features are 1 for the presence of certain words in the URL, alt text, or caption of the image. These words, such as the word sponsor, are used to determine if the image is likely to be an advertisement. Many of the features overlap considerably, as they are combinations of other features. Therefore, this dataset has a lot of redundant information.

With our dataset loaded in `pandas`, we will now extract the `x` and `y` data for our classification algorithms. The `x` matrix will be all of the columns in our Dataframe, except for the last column. In contrast, the `y` array will be only that last column, feature `1558`. Before that though, we simplify our dataset (just for this chapter's sake) by dropping any row with a NaN value. Let's look at the code:

```
ads.dropna(inplace=True)
X = ads.drop(1558, axis=1).values
y = ads[1558]
```

More than 1000 rows are dropped due to this command, which is fine for our exercise. For real-world applications, you don't want to discard data if you can help it--instead, you can use interpolation or value replacing to fill the NaN values. As an example, you can replace any missing value with the average for that column.

Principal Component Analysis

In some datasets, features heavily correlate with each other. For example, the speed and the fuel consumption would be heavily correlated in a go-kart with a single gear. While it can be useful to find these correlations for some applications, data mining algorithms typically do not need the redundant information.

The ads dataset has heavily correlated features, as many of the keywords are repeated across the alt text and caption.

The Principal Component Analysis (PCA) algorithm aims to find combinations of features that describe the dataset in less information. It aims to discover *principal components*, which are features that do not correlate with each other and explain the information—specifically the variance—of the dataset. What this means is that we can often capture most of the information in a dataset in fewer features.

We apply PCA just like any other transformer. It has one key parameter, which is the number of components to find. By default, it will result in as many features as you have in the original dataset. However, these principal components are ranked—the first feature explains the largest amount of the variance in the dataset, the second a little less, and so on. Therefore, finding just the first few features is often enough to explain much of the dataset. Let's look at the code:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=5)
Xd = pca.fit_transform(X)
```

The resulting matrix, **Xd**, has just five features. However, let's look at the amount of variance that is explained by each of these features:

```
np.set_printoptions(precision=3, suppress=True)
pca.explained_variance_ratio_
```

The result, `array([ 0.854, 0.145, 0.001, 0. , 0. ])`, shows us that the first feature accounts for 85.4 percent of the variance in the dataset, the second accounts for 14.5 percent, and so on. By the fourth feature, less than one-tenth of a percent of the variance is contained in the feature. The other 1,553 features explain even less (this is an ordered array).

The downside to transforming data with PCA is that these features are often complex combinations of the other features. For example, the first feature of the preceding code starts with `[-0.092, -0.995, -0.024]`, that is, multiply the first feature in the original dataset by -0.092, the second by -0.995, the third by -0.024. This feature has 1,558 values of this form, one for each of the original datasets (although many are zeros). Such features are indistinguishable by humans and it is hard to glean much relevant information from without a lot of experience working with them.

Using PCA can result in models that not only approximate the original dataset, but can also improve the performance in classification tasks:

```
clf = DecisionTreeClassifier(random_state=14)
scores_reduced = cross_val_score(clf, Xd, y, scoring='accuracy')
```

The resulting score is 0.9356, which is (slightly) higher than our original model's score. PCA won't always give a benefit like this, but it does more often than not.

We are using PCA here to reduce the number of features in our dataset. As a general rule, you shouldn't use it to reduce overfitting in your data mining experiments. The reason for this is that PCA doesn't take classes into account. A better solution is to use regularization. An introduction, with code, is available at `http://blog.datadive.net/selecting-good-features-part-ii-linear-models-and-regularization/`

Another advantage is that PCA allows you to plot datasets that you otherwise couldn't easily visualize. For example, we can plot the first two features returned by PCA.

First, we tell our Notebook to display plots inline:

```
%matplotlib inline
from matplotlib import pyplot as plt
```

Next, we get all of the distinct classes in our dataset (there are only two: is ad or not ad):

```
classes = set(y)
```

We also assign colors to each of these classes:

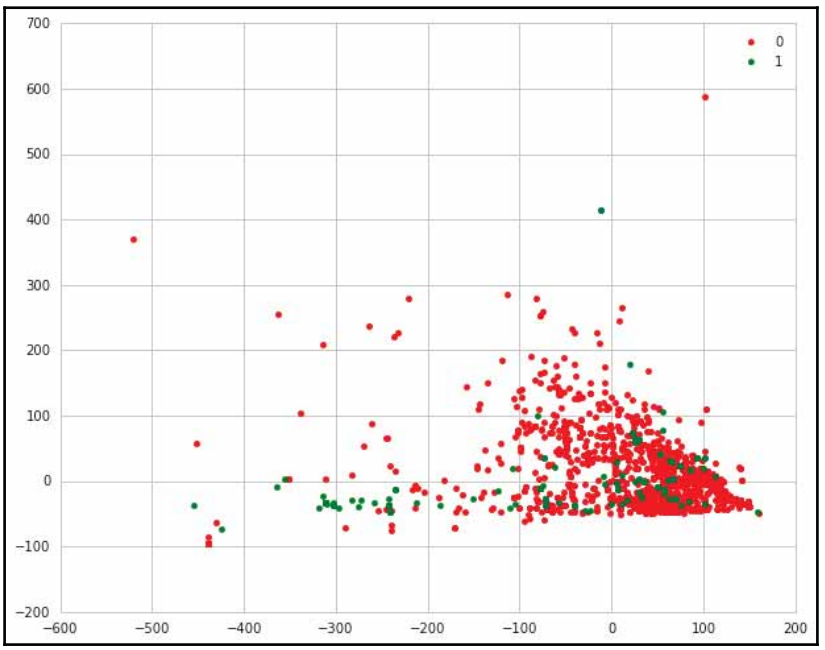
```
colors = ['red', 'green']
```

We use zip to iterate over both lists at the same time, then extract all samples from that class, and plot them with the color appropriate to the class:

```
for cur_class, color in zip(classes, colors):
mask = (y == cur_class)
    plt.scatter(Xd[mask,0], Xd[mask,1], marker='o', color=color,
label=int(cur_class))
```

Finally, outside the loop, we create a legend and show the graph, showing where the samples from each class appear:

```
plt.legend()
plt.show()
```



Creating your own transformer

As the complexity and type of dataset changes, you might find that you can't find an existing feature extraction transformer that fits your needs. We will see an example of this in `Chapter 7`, *Follow Recommendations Using Graph Mining*, where we create new features from graphs.

A transformer is akin to a converting function. It takes data of one form as input and returns data of another form as output. Transformers can be trained using some training dataset, and these trained parameters can be used to convert testing data.

The transformer API is quite simple. It takes data of a specific format as input and returns data of another format (either the same as the input or different) as output. Not much else is required of the programmer.

The transformer API

Transformers have two key functions:

- `fit()`: This takes a training set of data as input and sets internal parameters
- `transform()`: This performs the transformation itself. This can take either the training dataset, or a new dataset of the same format

Both `fit()` and `transform()` functions should take the same data type as input, but `transform()` can return data of a different type while `fit()` always returns **self**.

We are going to create a trivial transformer to show the API in action. The transformer will take a NumPy array as input, and discretize it based on the mean. Any value higher than the mean (of the training data) will be given the value 1 and any value lower or equal to the mean will be given the value 0.

We did a similar transformation with the Adult dataset using pandas: we took the **Hours-per-week feature** and created a **LongHours** feature if the value was more than 40 hours per week. This transformer is different for two reasons. First, the code will conform to the **scikit-learn** API, allowing us to use it in a pipeline. Second, the code will learn the mean, rather than taking it as a fixed value (such as 40 in the **LongHours** example).

Implementing a Transformer

To start, open up the Jupyter Notebook that we used for the Adult dataset. Then, click on the **Cell** menu item and choose **Run All**. This will rerun all of the cells and ensure that the notebook is up to date.

First, we import the **TransformerMixin**, which sets the API for us. While Python doesn't have strict interfaces (as opposed to languages like Java), using a mixin like this allows **scikit-learn** to determine that the class is actually a transformer. We also need to import a function that checks the input is of a valid type. We will use that soon.

Let's look at the code:

```
from sklearn.base import TransformerMixin
from sklearn.utils import as_float_array
```

Let's take a look at our class in entirety, and then we will revisit some of the details:

```
class MeanDiscrete(TransformerMixin):
    def fit(self, X, y=None):
        X = as_float_array(X)
        self.mean = X.mean(axis=0)
        return self

    def transform(self, X, y=None):
        X = as_float_array(X)
        assert X.shape[1] == self.mean.shape[0]
        return X > self.mean
```

Our class will learn the mean for each feature in the fit method, by computing `X.mean(axis=0)`, which is then stored as an object attribute. After that, the **fit** function returns **self**, conforming to the API (scikit-learn uses this to allow for chaining function calls).

After fitting, the transform function takes a matrix with the same number of features (confirmed by the `assert` statement), and simply returns which values are more than the mean for a given feature.

Now that our class is built, we can now create an instance of this class and use it to transform our **X** array:

```
mean_discrete = MeanDiscrete()
X_mean = mean_discrete.fit_transform(X)
```

Take a shot at implementing this Transformer into a workflow, both using a Pipeline and without. You'll see that by conforming to the Transformer API, it is quite simple to use in place of a built-in scikit-learn Transformer object.

Unit testing

When creating your own functions and classes, it is always a good idea to do unit testing. Unit testing aims to test a single unit of your code. In this case, we want to test that our transformer does as it needs to do.

Good tests should be independently verifiable. A good way to confirm the legitimacy of your tests is by using another computer language or method to perform the calculations. In this case, I used Excel to create a dataset, and then computed the mean for each cell. Those values were then transferred to the unit test.

Unit tests should also, generally, be small and quick to run. Therefore, any data used should be of a small size. The dataset I used for creating the tests is stored in the **Xt** variable from earlier, which we will recreate in our test. The mean of these two features is 13.5 and 15.5, respectively.

To create our unit test, we import the `assert_array_equal` function from **NumPy's** testing, which checks whether two arrays are equal:

```
from numpy.testing import assert_array_equal
```

Next, we create our function. It is important that the test's name starts with test_, as this nomenclature is used for tools that automatically find and run tests. We also set up our testing data:

```
def test_meandiscrete():
    X_test = np.array([[ 0, 2],
                       [ 3, 5],
                       [ 6, 8],
                       [ 9, 11],
                       [12, 14],
                       [15, 17],
                       [18, 20],
                       [21, 23],
                       [24, 26],
                       [27, 29]])
    # Create an instance of our Transformer
    mean_discrete = MeanDiscrete()
    mean_discrete.fit(X_test)
    # Check that the computed mean is correct
```

```
    assert_array_equal(mean_discrete.mean, np.array([13.5, 15.5]))
    # Also test that transform works properly
    X_transformed = mean_discrete.transform(X_test)
    X_expected = np.array([[ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1]])
    assert_array_equal(X_transformed, X_expected)
```

We can run the test by simply running the function itself:

```
test_meandiscrete()
```

If there was no error, then the test ran without an issue! You can verify this by changing some of the tests to deliberately make values incorrect, and confirming that the test fails. Remember to change them back so that the test passes!

If we had multiple tests, it would be worth using a testing framework, like **py.test** or **nose** to run our tests. Using a framework like this is beyond the scope of this book, but they manage running tests, recording failures, and providing feedback to you, as a programmer, to help you improve your code.

Putting it all together

Now that we have a tested transformer, it is time to put it into action. Using what we have learned so far, we create a Pipeline, set the first step to the **MeanDiscrete** transformer, and the second step to a Decision Tree Classifier. We then run a cross-validation and print out the result. Let's look at the code:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('mean_discrete', MeanDiscrete()), ('classifier',
DecisionTreeClassifier(random_state=14))])
scores_mean_discrete = cross_val_score(pipeline, X, y, scoring='accuracy')
print("Mean Discrete performance:
{0:.3f}".format(scores_mean_discrete.mean()))
```

The result is 0.917, which is not as good as before, but very good for a simple binary feature model.

Summary

In this chapter, we looked at features and transformers and how they can be used in the data mining pipeline. We discussed what makes a good feature and how to algorithmically choose good features from a standard set. However, creating good features is more art than science and often requires domain knowledge and experience.

We then created our own transformer using an interface that allows us to use it in scikit-learn's helper functions. We will be creating more transformers in later chapters so that we can perform effective testing using existing functions.

To take the lessons learned in this chapter further, I recommend signing up to the online data mining competition website `Kaggle.com` and trying some of the competitions. Their recommended starting place is the Titanic dataset, which allows you to practice the feature creation aspects of this chapter. Many of the features are not numerical, requiring you to convert them to numerical features before applying a data mining algorithm.

In the next chapter, we use feature extraction on a corpus of text documents. There are many transformers and feature types for text, each with their advantages and disadvantages.

6

Social Media Insight using Naive Bayes

Text-based documents contain lots of information. Examples include books, legal documents, social media, and e-mail. Extracting information from text-based documents is critically important to modern AI systems, for example in search engines, legal AI, and automated news services.

Extraction of useful features from text is a difficult problem. Text is not numerical in nature, therefore a model must be used to create features that can be used with data mining algorithms. The good news is that there are some simple models that do a great job at this, including the bag-of-words model that we will use in this chapter.

In this chapter, we look at extracting features from text for use in data mining applications. The specific problem we tackle in this chapter is term disambiguation on social media - determining which meaning a word has based on its context.

We will cover the following topics in this chapter:

- Downloading data from social network APIs
- Transformers and models for text data
- The Naive Bayes classifier
- Using JSON for saving and loading datasets
- The NLTK library for feature creation
- The F-measure for evaluation

Disambiguation

Text data is often called an *unstructured format*. There is a lot of information in text, but it is just *there*; no headings, no required format (save for normal grammatical rules), loose syntax, and other problems prohibit the easy extraction of information from text. The data is also highly connected, with lots of mentions and cross-references—just not in a format that allows us to easily extract it! Even seemingly easy problems, such as determining if a word is a noun, have lots of weird edge cases that make it difficult to do reliably.

We can compare the information stored in a book with that stored in a large database to see the difference. In the book, there are characters, themes, places, and lots of information. However, a book needs to be read and interpreted, with cultural context, to gain this information. In contrast, a database sits on your server with column names and data types. All the information is there and the level of interpretation needed to extract specific information is quite low.

Information about the data, such as its type or its meaning, is called metadata. Text lacks metadata. A book also contains some metadata in the form of a table of contents and index but the degree of information included in these sections is significantly lower than that of a database.

One of the problems in working with text is **term disambiguation**. When a person uses the word *bank*, is this a financial message or an environmental message (such as river bank)? This type of disambiguation is quite easy in many circumstances for humans (although there are still troubles), but much harder for computers to do.

In this chapter, we will look at disambiguating the use of the term **Python** on Twitter's stream. When people talk about Python, they could be talking about the following things:

- The programming language Python
- Monty Python, the classic comedy group
- The snake Python
- A make of shoe called Python

There can be many other things called Python. The aim of our experiment is to take a tweet mentioning Python and determine whether it is talking about the programming language, based only on the content of the tweet.

A message on Twitter is called a *tweet* and is limited to 140 characters. Tweets include lots of metadata, such as the time and date of posting, who posted it, and so on. However in regards to the topic of the tweet, there is not much in this regard.

In this chapter, we are going to perform a data mining experiment consisting of the following steps:

1. Download a set of tweets from Twitter.
2. Manually classify them to create a dataset.
3. Save the dataset so that we can replicate our research.
4. Use the Naive Bayes classifier to create a classifier to perform term disambiguation.

Downloading data from a social network

We are first going to download a corpus of data from Twitter and use it to sort out spam from useful content. Twitter provides a robust API for collecting information from its servers and this API is free for small-scale usage. It is, however, subject to some conditions that you'll need to be aware of if you start using Twitter's data in a commercial setting.

First, you'll need to sign up for a Twitter account (which is free). Go to `http://twitter.com` and register an account if you do not already have one.

Next, you'll need to ensure that you only make a certain number of requests per minute. This limit is currently 15 requests per 15 minutes (it depends on the exact API). It can be tricky ensuring that you don't breach this limit, so it is highly recommended that you use a library to talk to Twitter's API.

If you are using your own code (that is making the web calls with your own code) to connect with a web-based API, ensure that you read the documentation about rate limiting their documentation and understand the limitations. In Python, you can use the `time` library to perform a pause between calls to ensure you do not breach the limit.

You will then need a key to access Twitter's data. Go to `http://twitter.com`and sign in to your account. When you are logged in, go to `https://apps.twitter.com/` and click on **Create New App**. Create a name and description for your app, along with a website address.

If you don't have a website to use, insert a placeholder. Leave the **Callback URL** field blank for this app—we won't need it. Agree to the terms of use (if you do) and click on **Create your Twitter application**.

Keep the resulting website open—you'll need the access keys that are on this page. Next, we need a library to talk to Twitter. There are many options; the one I like is simply called `twitter`, and is the official Twitter Python library.

You can install `twitter` using **pip3 install twitter** (on the command line) if you are using pip to install your packages. At the time of writing, Anaconda does not include twitter, therefore you can't use `conda` to install it. If you are using another system or want to build from source, check the documentation at `https://github.com/sixohsix/twitter`

Create a new Jupyter Notebook to download the data. We will create several notebooks in this chapter for various different purposes, so it might be a good idea to also create a folder to keep track of them. This first notebook, `ch6_get_twitter`, is specifically for downloading new Twitter data.

First, we import the twitter library and set our authorization tokens. The consumer key and consumer secret will be available on the **Keys and Access Tokens** tab on your Twitter app's page. To get the access tokens, you'll need to click on the Create my access token button, which is on the same page. Enter the keys into the appropriate places in the following code:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
consumer_key, consumer_secret)
```

We are going to get our tweets from Twitter's search function. We will create a reader that connects to twitter using our authorization, and then use that reader to perform searches. In the Notebook, we set the filename where the tweets will be stored:

```
import os
output_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_tweets.json")
```

Next, create an object that can read from Twitter. We create this object with our authorization object that we set up earlier:

```
t = twitter.Twitter(auth=authorization)
```

We then open our output file for writing. We open it for appending—this allows us to rerun the script to obtain more tweets. We then use our Twitter connection to perform a search for the word Python. We only want the statuses that are returned for our dataset. This code takes the tweet, uses the **json** library to create a string representation using the dumps function, and then writes it to the file. It then creates a blank line under the tweet so that we can easily distinguish where one tweet starts and ends in our file:

```
import json
with open(output_filename, 'a') as output_file:
    search_results = t.search.tweets(q="python", count=100)['statuses']
    for tweet in search_results:
        if 'text' in tweet:
            output_file.write(json.dumps(tweet))
            output_file.write("nn")
```

In the preceding loop, we also perform a check to see whether there is text in the tweet or not. Not all of the objects returned by twitter will be actual tweets (for example, some responses will be actions to delete tweets). The key difference is the inclusion of text as a key, which we test for. Running this for a few minutes will result in 100 tweets being added to the output file.

You can keep re-running this script to add more tweets to your dataset, keeping in mind that you may get some duplicates in the output file if you rerun it too fast (that is before Twitter gets new tweets to return!). For our initial experiment, 100 tweets will be enough, but you will probably want to come back and rerun this code to get that up to about 1000.

Loading and classifying the dataset

After we have collected a set of tweets (our dataset), we need labels to perform classification. We are going to label the dataset by setting up a form in a Jupyter Notebook to allow us to enter the labels. We do this by loading the tweets we collected in the previous section, iterating over them and providing (manually) a classification on whether they refer to Python the programming language or not.

The dataset we have stored is nearly, but not quite, in a **JSON** format. JSON is a format for data that doesn't impose much structure on the contents, just on the syntax. The idea behind JSON is that the data is in a format directly readable in JavaScript (hence the name, *JavaScript Object Notation*). JSON defines basic objects such as numbers, strings, lists, and dictionaries, making it a good format for storing datasets, if they contain data that isn't numerical. If your dataset is fully numerical, you would save space and time using a matrix-based format like in NumPy.

A key difference between our dataset and real JSON is that we included newlines between tweets. The reason for this was to allow us to easily append new tweets (the actual JSON format doesn't allow this easily). Our format is a JSON representation of a Tweet, followed by a newline, followed by the next Tweet, and so on.

To parse it, we can use the **json** library but we will have to first split the file by newlines to get the actual tweet objects themselves. Set up a new Jupyter Notebook, I called mine **ch6_label_twitter**. Within it, we will first load the data from our input filename by iterating over the file, storing tweets as we loop. The code below does a basic check that there is actual text in the tweet. If it does, we use the **json** library to load the tweet and then we add it to a list:

```
import json
import os

# Input filename
input_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_tweets.json")
# Output filename
labels_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_classes.json")

tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
```

We are now interested in manually classifying whether an item is relevant to us or not (in this case, relevant means refers to the programming language Python). We will use the Jupyter Notebook's ability to embed HTML and talk between JavaScript and Python to create a viewer of tweets to allow us to easily and quickly classify the tweets as spam or not. The code will present a new tweet to the user (you) and ask for a label: *is it relevant or not?* It will then store the input and present the next tweet to be labeled.

First, we create a list for storing the labels. These labels will be stored whether or not the given tweet refers to the programming language Python, and it will allow our classifier to learn how to differentiate between meanings.

We also check if we have any labels already and load them. This helps if you need to close the notebook down midway through labeling. This code will load the labels from where you left off. It is generally a good idea to consider how to save at midpoints for tasks like this. Nothing hurts quite like losing an hour of work because your computer crashed before you saved the labels! The code to do this loading follows:

```
labels = []
if os.path.exists(labels_filename):
    with open(labels_filename) as inf:
        labels = json.load(inf)
```

The first time you run this, nothing will happen. After manually classifying some examples you can save your progress and then close the Notebook. After that, you can reopen the Notebook and return to where you were up to.

If you make one or two mistakes classifying, don't worry too much. If you make lots of mistakes and want to start again, delete just **python_classes.json** and the above code will pick up with an empty set of classifications. If you need to delete all of your data and start again with new tweets, make sure to delete (or move) both files - **python_tweets.json** and **python_classes.json**. Otherwise, this Notebook will get confused, giving classes from the old dataset to the new tweets.

Next, we create a simple function that will return the next tweet that needs to be labeled. We can work out which is the next tweet by finding the first one that hasn't yet been labeled. The code is pretty straight-forward. We determine how many tweets we have labeled (with `len(labels)`), and get the next tweet in the **tweet_sample** list:

```
def get_next_tweet():
    return tweets[len(labels)]['text']
```

The next step in our experiment is to collect information from the user (you!) on which tweets are referring to Python (the programming language) and which are not.

As of yet, there is not a good, straightforward way to get interactive feedback with pure Python in Jupyter Notebooks for such a large number of text documents. For this reason, we will use some JavaScript and HTML to get this input from the user. There are many ways to do this, below is just one example.

To get the feedback, we need a JavaScript component to load the next tweet and show it. We also need a HTML component to create the HTML elements to display that tweet. I won't go into the details of the code here, except to give this general workflow:

1. Obtain the next tweet that needs to be classified with `load_next_tweet`
2. Show it to the user with `handle_output`
3. Wait for the user to press either 0 or 1 with `$("input#capture").keypress`
4. Store that result in the classes list with `set_label`

This keeps happening until we reach the end of the list (at which point an IndexError occurs, indicating we have no more tweets to classify). The code is below (remember that you can get the code from Packt or from the official GitHub repository):

```
%%html
<div name="tweetbox">
 Instructions: Click in text box. Enter a 1 if the tweet is relevant, enter
0 otherwise.<br>
 Tweet: <div id="tweet_text" value="text"></div><br>
 <input type=text id="capture"></input><br>
</div>

<script>
function set_label(label){
 var kernel = IPython.notebook.kernel;
 kernel.execute("labels.append(" + label + ")");
 load_next_tweet();
}

function load_next_tweet(){
 var code_input = "get_next_tweet()";
 var kernel = IPython.notebook.kernel;
 var callbacks = { 'iopub' : {'output' : handle_output}};
 kernel.execute(code_input, callbacks, {silent:false});
}

function handle_output(out){
 console.log(out);
 var res = out.content.data["text/plain"];
 $("div#tweet_text").html(res);
}

$("input#capture").keypress(function(e) {
 console.log(e);
 if(e.which == 48) {
 // 0 pressed
 set_label(0);
```

```
 $("input#capture").val("");
 }else if (e.which == 49){
 // 1 pressed
 set_label(1);
 $("input#capture").val("");
 }
});

load_next_tweet();
</script>
```

You will need to enter all of this code into a single cell (or copy it from the code bundle). It contains the mix of HTML and JavaScript necessary to get input from you to manually classify the tweets. If you need to stop or save your progress, run the following code in the next cell. It will save your progress (and doesn't interrupt the above HTML code either, which can be left running):

```
with open(labels_filename, 'w') as outf:
    json.dump(labels, outf)
```

Creating a replicable dataset from Twitter

In data mining, there are lots of variables. These aren't the parameters of the data mining algorithms - they are the methods of data collection, how the environment is set up, and many other factors. Being able to replicate your results is important as it enables you to verify or improve upon your results.

Getting 80 percent accuracy on one dataset with algorithm X, and 90 percent accuracy on another dataset with algorithm Y doesn't mean that Y is better. We need to be able to test on the same dataset in the same conditions to be able to properly compare. With running the preceding code, you will get a different dataset to the one I created and used. The main reasons are that Twitter will return different search results for you than me based on the time you performed the search.

Even after that, your labeling of tweets might be different from what I do. While there are obvious examples where a given tweet relates to the python programming language, there will always be gray areas where the labeling isn't obvious. One tough gray area I ran into was tweets in non-English languages that I couldn't read. In this specific instance, there are options in Twitter's API for setting the language, but even these aren't going to be perfect.

Due to these factors, it is difficult to replicate experiments on databases that are extracted from social media, and Twitter is no exception. Twitter explicitly disallows sharing datasets directly. One solution to this is to share tweet IDs only, which you can share freely. In this section, we will first create a tweet ID dataset that we can freely share. Then, we will see how to download the original tweets from this file to recreate the original dataset. First, we save the replicable dataset of tweet IDs.

After creating another new Jupyter Notebook, first set up the filenames as before. This is done in the same way we did labeling but there is a new filename where we can store the replicable dataset. The code is as follows:

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "replicable_dataset.json")
```

We load the tweets and labels as we did in the previous notebook:

```
import json
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
if os.path.exists(labels_filename):
    with open(labels_filename) as inf:
        labels = json.load(inf)
```

Now we create a dataset by looping over both the tweets and labels at the same time and saving those in a list. An important side-effect of this code is, by putting labels first in the zip function, it will only load enough tweets for the labels we have created. In other words, you can run this code on partially classified data:

```
dataset = [(tweet['id'], label) for label, tweet in zip(labels, tweets)]
```

Finally, we save the results in our file:

```
with open(replicable_dataset, 'w') as outf:
    json.dump(dataset, outf)
```

Now that we have the Tweet IDs and labels saved, we can recreate the original dataset. If you are looking to recreate the dataset I used for this chapter, it can be found in the code bundle that comes with this book. Loading the preceding dataset is not difficult but it can take some time.

Start a new Jupyter Notebook and set the dataset, label, and tweet ID filenames as before. I've adjusted the filenames here to ensure that you don't overwrite your previously collected dataset, but feel free to change these if you do want to override.

The code is as follows:

```
import os
tweet_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"replicable_python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"replicable_python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "replicable_dataset.json")
```

Next, load the tweet IDs from the file using JSON:

```
import json
with open(replicable_dataset) as inf:
    tweet_ids = json.load(inf)
```

Saving the labels is very easy. We just iterate through this dataset and extract the IDs. We could do this quite easily with just two lines of code (open file and save tweets). However, we can't guarantee that we will get all the tweets we are after (for example, some may have been changed to private since collecting the dataset) and therefore the labels will be incorrectly indexed against the data. As an example, I tried to recreate the dataset just one day after collecting them and already two of the tweets were missing (they might be deleted or made private by the user). For this reason, it is important to only print out the labels that we need.

To do this, we first create an empty actual labels list to store the labels for tweets that we actually recover from twitter, and then create a dictionary mapping the tweet IDs to the labels. The code is as follows:

```
actual_labels = []
label_mapping = dict(tweet_ids)
```

Next, we are going to create a twitter server to collect all of these tweets. This is going to take a little longer. Import the twitter library that we used before, creating an authorization token and using that to create the twitter object:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization)
```

Next, we will loop through each of the tweet ids, and ask twitter to recover the original tweet. A good feature of twitter's API is that we can ask for 100 tweets at a time, drastically reducing the number of API calls. Interestingly, from twitter's point of view, it is the same number of calls to get one tweet or 100 tweets, as long as its a single request.

The following code will loop through our tweets in groups of 100, join together the id values, and get the tweet information for each of them.

```
all_ids = [tweet_id for tweet_id, label in tweet_ids]
with open(tweet_filename, 'a') as output_file:
    # We can lookup 100 tweets at a time, which saves time in asking
twitter for them
    for start_index in range(0, len(all_ids), 100):
        id_string = ",".join(str(i) for i in
all_ids[start_index:start_index+100])
        search_results = t.statuses.lookup(_id=id_string)
        for tweet in search_results:
            if 'text' in tweet:
                # Valid tweet - save to file
                output_file.write(json.dumps(tweet))
                output_file.write("nn")
                actual_labels.append(label_mapping[tweet['id']])
```

In this code, we then check each tweet to see if it is a valid tweet and then save it to our file if it is. Our final step is to save our resulting labels:

```
with open(labels_filename, 'w') as outf:
    json.dump(actual_labels, outf)
```

Text transformers

Now that we have our dataset, how are we going to perform data mining on it?

Text-based datasets include books, essays, websites, manuscripts, programming code, and other forms of written expression. All of the algorithms we have seen so far deal with numerical or categorical features, so how do we convert our text into a format that the algorithm can deal with? There are a number of measurements that could be taken.

For instance, average word and average sentence length are used to predict the readability of a document. However, there are lots of feature types such as word occurrence which we will now investigate.

Bag-of-words models

One of the simplest but highly effective models is to simply count each word in the dataset. We create a matrix, where each row represents a document in our dataset and each column represents a word. The value of the cell is the frequency of that word in the document. This is known as the **bag-of-words model**.

Here's an excerpt from *The Lord of the Rings, J.R.R. Tolkien:*

Three Rings for the Elven-kings under the sky,

Seven for the Dwarf-lords in halls of stone, Nine for Mortal Men, doomed to die,

One for the Dark Lord on his dark throne In the Land of Mordor where the Shadows lie.

One Ring to rule them all, One Ring to find them,

One Ring to bring them all and in the darkness bind them.

In the Land of Mordor where the Shadows lie.

- J.R.R. Tolkien's epigraph to The Lord of The Rings

The word **the** appears nine times in this quote, while the words **in**, **for**, **to**, and **one** each appear four times. The word **ring** appears three times, as does the word **of**.

We can create a dataset from this, choosing a subset of words and counting the frequency:

Word	**the**	**one**	**ring**	**to**
Frequency	9	4	3	4

To do this for all words in a single document, we can use the **Counter** class. When counting words, it is normal to convert all letters to lowercase, which we do when creating the string. The code is as follows:

```
 s = """Three Rings for the Elven-kings under the sky, Seven for the Dwarf-
lords in halls of stone, Nine for Mortal Men, doomed to die, One for the
Dark Lord on his dark throne In the Land of Mordor where the Shadows lie.
One Ring to rule them all, One Ring to find them, One Ring to bring them
all and in the darkness bind them. In the Land of Mordor where the Shadows
lie. """.lower()
words = s.split()
from collections import Counter
c = Counter(words)
print(c.most_common(5))
```

Printing `c.most_common(5)` gives the list of the top five most frequently occurring words. Ties are not handled well as only five are given and a very large number of words all share a tie for fifth place.

The bag-of-words model has three major types, with many variations and alterations.

- The first is to use the raw frequencies, as shown in the preceding example. This has the same drawback as any non-normalised data - words with high variance due to high overall values (such as) *the* overshadow lower frequency (and therefore lower-variance) words, even though the presence of the word *the* rarely has much importance.
- The second model is to use the normalized frequency, where each document's sum equals 1. This is a much better solution as the length of the document doesn't matter as much, but it still means words like the overshadow lower frequency words. The third type is to simply use binary features—a value is 1 if it occurs, and 0 otherwise. We will use binary representation in this chapter.
- Another (arguably more popular) method for performing normalization is called **term frequency-inverse document frequency** (**tf-idf**). In this weighting scheme, term counts are first normalized to frequencies and then divided by the number of documents in which it appears in the corpus. We will use tf-idf in `Chapter 10`, *Clustering News Articles*.

n-gram features

One variation on the standard bag-of-words model is called the **n-gram** model. An n-grams model addresses the deficiency of context in the bag-of-words model. With a bag-of-words model, only individual words are counted by themselves. This means that common word pairs, such as *United States*, lose meaning they have in the sentence because they are treated as individual words.

There are algorithms that can read a sentence, parse it into a tree-like structure, and use this to create very accurate representations of the meaning behind words. Unfortunately, these algorithms are computationally expensive. This makes it difficult to apply them to large datasets.

To compensate for these issues of context and complexity, the n-grams model fits into the middle ground. It has more context than the bag-of-words model, while only being slightly more expensive computationally.

An n-gram is a subsequence of *n* consecutive, overlapping, tokens. In this experiment, we use word n-grams, which are n-grams of word-tokens. They are counted the same way as a bag-of-words, with the n-grams forming a *word* that is put in the bag. The value of a cell in this dataset is the frequency that a particular n-gram appears in the given document.

The value of n is a parameter. For English, setting it to between 2 to 5 is a good start, although some applications call for higher values. Higher values for n result in sparse datasets, as when n increases it is less likely to have the same n-gram appear across multiple documents. Having n=1 results in simply the bag-of-words model.

As an example, `for n=3`, we extract the first few n-grams in the following quote:

> *Always look on the bright side of life.*

The first n-gram (of size 3) is *Always look on*, the second is *look on the*, the third is *on the bright*. As you can see, the n-grams overlap and cover three words each. Word n-grams have advantages over using single words. This simple concept introduces some context to word use by considering its local environment, without a large overhead of understanding the language computationally.

A disadvantage of using n-grams is that the matrix becomes even sparser—word n-grams are unlikely to appear twice (especially in tweets and other short documents!). Specifically for social media and other short documents, word n-grams are unlikely to appear in too many different tweets, unless it is a retweet. However, in larger documents, word n-grams are quite effective for many applications. Another form of n-gram for text documents is that of a character n-gram. That said, you'll see shortly that word n-grams are quite effective in practice.

Rather than using sets of words, we simply use sets of characters (although character n-grams have lots of options for how they are computed!). This type of model can help identify words that are misspelled, as well as providing other benefits to classification. We will test character n-grams in this chapter and see them again in `Chapter 9`, *Authorship Attribution*.

Other text features

There are other features that can be extracted too. These include syntactic features, such as the usage of particular words in sentences. Part-of-speech tags are also popular for data mining applications that need to understand meaning in text. Such feature types won't be covered in this book. If you are interested in learning more, I recommend *Python 3 Text Processing with NLTK 3 Cookbook, Jacob Perkins, Packt publication*.

There are a number of libraries for working with text data in Python. The most commonly known one is called the **Natural Language ToolKit (NLTK)**. The **scikit-learn** library also has the **CountVectorizer** class that performs a similar action, and it is recommended you take a look at it (we will use it in `Chapter 9`, *Authorship Attribution*). NLTK has more features for word tokenization and part of speech tagging (that is identifying which words are nouns, verbs and so on).

The library we are going to use is called spaCy. It is designed from the ground up to be fast and reliable for natural language processing. Its less-well-known than NLTK, but is rapidly growing in popularity. It also simplifies some of the decisions, but has a slightly more difficult syntax to use, compared to NLTK.

For production systems, I recommend using spaCy, which is faster than NLTK. NLTK was built for teaching, while spaCy was built for production. They have different syntaxes, meaning it can be difficult to port code from one library to another. If you aren't looking into experimenting with different types of natural language parsers, I recommend using spaCy.

Naive Bayes

Naive Bayes is a probabilistic model that is, unsurprisingly, built upon a naive interpretation of Bayesian statistics. Despite the naive aspect, the method performs very well in a large number of contexts. Because of the naive aspect, it works quite quickly. It can be used for classification of many different feature types and formats, but we will focus on one in this chapter: binary features in the bag-of-words model.

Understanding Bayes' theorem

For most of us, when we were taught statistics, we started from a **frequentist approach**. In this approach, we assume the data comes from some distribution and we aim to determine what the parameters are for that distribution. However, those parameters are (perhaps incorrectly) assumed to be fixed. We use our model to describe the data, even testing to ensure the data fits our model.

Bayesian statistics instead model how people (at least, non-frequentist statisticians) actually reason. We have some data, and we use that data to update our model about how likely something is to occur. In Bayesian statistics, we use the data to describe the model rather than using a model and confirming it with data (as per the frequentist approach).

It should be noted that frequentist statistics and Bayesian statistics ask and answer slightly different questions. A direct comparison is not always correct.

Bayes' theorem computes the value of **P(A|B)**. That is, knowing that **B** has occurred, what is the probability of event **A** occurring. In most cases, **B** is an observed event such as *it rained yesterday*, and **A** is a prediction it will rain today. For data mining, **B** is usually *we observed this sample* and **A** is *does the sample belong to this class* (the class prediction). We will see how to use Bayes' theorem for data mining in the next section.

The equation for Bayes' theorem is given as follows:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}.$$

As an example, we want to determine the probability that an e-mail containing the word **drugs** is spam (as we believe that such a tweet may be a pharmaceutical spam).

A, in this context, is the probability that this tweet is spam. We can compute *P(A)*, called the prior belief directly from a training data by computing the percentage of tweets in our dataset that are spam. If our dataset contains 30 spam messages for every 100 e-mails, *P(A)* is 30/100 or 0.3.

B, in this context, is this tweet contains the word *drugs*. Likewise, we can compute *P(B)* by computing the percentage of tweets in our dataset containing the word drugs. If 10 e-mails in every 100 of our training dataset contain the word drugs, *P(B)* is 10/100 or 0.1. Note that we don't care if the e-mail is spam or not when computing this value.

P(B|A) is the probability that an e-mail contains the word drugs if it is spam. This is also easy to compute from our training dataset. We look through our training set for spam e-mails and compute the percentage of them that contain the word drugs. Of our 30 spam e-mails, if 6 contain the word drugs, then *P(B|A)* is calculated as 6/30 or 0.2.

From here, we use Bayes' theorem to compute *P(A|B)*, which is the probability that a tweet containing the word drugs is spam. Using the previous equation, we see the result is 0.6. This indicates that if an e-mail has the word drugs in it, there is a 60 percent chance that it is spam.

Note the empirical nature of the preceding example—we use evidence directly from our training dataset, not from some preconceived distribution. In contrast, a frequentist view of this would rely on us creating a distribution of the probability of words in tweets to compute similar equations.

Naive Bayes algorithm

Looking back at our Bayes' theorem equation, we can use it to compute the probability that a given sample belongs to a given class. This allows the equation to be used as a classification algorithm.

With *C* as a given class and *D* as a sample in our dataset, we create the elements necessary for Bayes' theorem, and subsequently Naive Bayes. Naive Bayes is a classification algorithm that utilizes Bayes' theorem to compute the probability that a new data sample belongs to a particular class.

P(D) is the probability of a given data sample. It can be difficult to compute this, as the sample is a complex interaction between different features, but luckily it is constant across all classes. Therefore, we don't need to compute it at all, as all we do in the final step is compare relative values.

P(D|C) is the probability of the data point belonging to the class. This could also be difficult to compute due to the different features. However, this is where we introduce the naive part of the Naive Bayes algorithm. We naively assume that each feature is independent of each other. Rather than computing the full probability of *P(D|C)*, we compute the probability of each feature *D1, D2, D3, ...* and so on. Then, we just multiply them together:

P*(D|C) = P(D1|C) x P(D2|C).... x P(Dn|C)*

Each of these values is relatively easy to compute with binary features; we simply compute the percentage of times it is equal in our sample dataset.

In contrast, if we were to perform a non-naive Bayes version of this part, we would need to compute the correlations between different features for each class. Such computation is infeasible at best, and nearly impossible without vast amounts of data or adequate language analysis models.

From here, the algorithm is straightforward. We compute **P(C|D)** for each possible class, ignoring the **P(D)** term entirely. Then we choose the class with the highest probability. As the **P(D)** term is consistent across each of the classes, ignoring it has no impact on the final prediction.

How it works

As an example, suppose we have the following (binary) feature values from a sample in our dataset: **[0, 0, 0, 1]**.

Our training dataset contains two classes with 75 percent of samples belonging to the class 0, and 25 percent belonging to the class 1. The likelihood of the feature values for each class are as follows:

For class 0: **[0.3, 0.4, 0.4, 0.7]**

For class 1: **[0.7, 0.3, 0.4, 0.9]**

These values are to be interpreted as: for feature 1, it has a value of 1 in 30 percent of cases for samples with class 0. It is a value of 1 in 70 percent of samples with class 1.

We can now compute the probability that this sample should belong to the class 0. **P(C=0) = 0.75** which is the probability that the class is 0. Again, **P(D)** isn't needed for the Naive Bayes algorithm and is simply removed from the equation. Let's take a look at the calculation:

```
P(D|C=0) = P(D1|C=0) x P(D2|C=0) x P(D3|C=0) x P(D4|C=0)
= 0.3 x 0.6 x 0.6 x 0.7
= 0.0756
```

The second and third values are 0.6, because the value of that feature in the sample was 0. The listed probabilities are for values of 1 for each feature. Therefore, the probability of a 0 is its inverse: *P(0) = 1 – P(1)*.

Now, we can compute the probability of the data point belonging to this class. Let's take a look at the calculation:

```
P(C=0|D) = P(C=0) P(D|C=0) = 0.75 * 0.0756 = 0.0567
```

Now, we compute the same values for the class 1:

```
P(D|C=1) = P(D1|C=1) x P(D2|C=1) x P(D3|C=1) x P(D4|C=1)
         = 0.7 x 0.7 x 0.6 x 0.9
         = 0.2646 P(C=1|D)
         = P(C=1)P(D|C=1)
         = 0.25 * 0.2646
         = 0.06615
```

Normally, **P(C=0|D) + P(C=1|D)** should equal to 1. After all, those are the only two possible options! However, the probabilities are not 1 due to the fact we haven't included the computation of **P(D)** in our equations here.

As the value for *P(C=1|D)* is more than *P(C=0|D)*, the data point should be classified as belonging to the class 1. You may have guessed this while going through the equations anyway; however, you may have been a bit surprised that the final decision was so close. After all, the probabilities in computing *P(D|C)* were much, much higher for the class 1. This is because we introduced a prior belief that most samples generally belong to the class 0.

If the classes had been equal sizes, the resulting probabilities would be much different. Try it yourself by changing both *P(C=0)* and *P(C=1)* to 0.5 for equal class sizes and computing the result again.

Applying of Naive Bayes

We will now create a pipeline that takes a tweet and determines whether it is relevant or not, based only on the content of that tweet.

To perform the word extraction, we will be using the spaCy, a library that contains a large number of tools for performing analysis on natural language. We will use spaCy in future chapters as well.

To get spaCy on your computer, use pip to install the package: **pip install spacy**
If that doesn't work, see the spaCy installation instructions at `https://spacy.io/` for information specific to your platform.

We are going to create a pipeline to extract the word features and classify the tweets using Naive Bayes. Our pipeline has the following steps:

- Transform the original text documents into a dictionary of counts using spaCy's word tokenization.
- Transform those dictionaries into a vector matrix using the **DictVectorizer** transformer in scikit-learn. This is necessary to enable the Naive Bayes classifier to read the feature values extracted in the first step.
- Train the Naive Bayes classifier, as we have seen in previous chapters.

We will need to create another Notebook (last one for the chapter!) called `ch6_classify_twitter` for performing the classification.

Extracting word counts

We are going to use spaCy to extract our word counts. We still want to use it in a pipeline, but spaCy doesn't conform to our transformer interface. We will need to create a basic transformer to do this to obtain both fit and transform methods, enabling us to use this in a pipeline.

First, set up the transformer class. We don't need to *fit* anything in this class, as this transformer simply extracts the words in the document. Therefore, our fit is an empty function, except that it returns self which is necessary for transformer objects to conform to the scikit-learn API.

Our transform is a little more complicated. We want to extract each word from each document and record True if it was discovered. We are only using the binary features here—True if in the document, False otherwise. If we wanted to use the frequency we would set up counting dictionaries, as we have done in several of the past chapters.

Let's take a look at the code:

```
import spacy
from sklearn.base import TransformerMixin

# Create a spaCy parser
nlp = spacy.load('en')


class BagOfWords(TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        results = []
        for document in X:
            row = {}
            for word in list(nlp(document, tag=False, parse=False,
entity=False)):
                if len(word.text.strip()): # Ignore words that are just
whitespace
                    row[word.text] = True
                    results.append(row)
        return results
```

The result is a list of dictionaries, where the first dictionary is the list of words in the first tweet, and so on. Each dictionary has a word as key and the value **True** to indicate this word was discovered. Any word not in the dictionary will be assumed to have not occurred in the tweet. Explicitly stating that a word's occurrence is **False** will also work, but will take up needless space to store.

Converting dictionaries to a matrix

The next step converts the dictionaries built as per the previous step into a matrix that can be used with a classifier. This step is made quite simple through the **DictVectorizer** transformer that is provided as part of scikit-learn.

The `DictVectorizer` class simply takes a list of dictionaries and converts them into a matrix. The features in this matrix are the keys in each of the dictionaries, and the values correspond to the occurrence of those features in each sample. Dictionaries are easy to create in code, but many data algorithm implementations prefer matrices. This makes `DictVectorizer` a very useful class.

In our dataset, each dictionary has words as keys and only occurs if the word actually occurs in the tweet. Therefore, our matrix will have each word as a feature and a value of True in the cell if the word occurred in the tweet.

To use `DictVectorizer`, simply import it using the following command:

```
from sklearn.feature_extraction import DictVectorizer
```

Putting it all together

Finally, we need to set up a classifier and we are using Naive Bayes for this chapter. As our dataset contains only binary features, we use the `BernoulliNB` classifier that is designed for binary features. As a classifier, it is very easy to use. As with `DictVectorizer`, we simply import it and add it to our pipeline:

```
from sklearn.naive_bayes import BernoulliNB
```

Now comes the moment to put all of these pieces together. In our Jupyter Notebook, set the filenames and load the dataset and classes as we have done before. Set the filenames for both the tweets themselves (not the IDs!) and the labels that we assigned to them. The code is as follows:

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data", "twitter",
"python_classes.json")
```

Load the tweets themselves. We are only interested in the content of the tweets, so we extract the text value and store only that. The code is as follows:

```
import json

tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0: continue
        tweets.append(json.loads(line)['text'])
```

```
with open(labels_filename) as inf:
    labels = json.load(inf)

# Ensure only classified tweets are loaded
tweets = tweets[:len(labels)]
```

Now, create a pipeline putting together the components from before. Our pipeline has three parts:

1. The NLTKBOW transformer we created.
2. A DictVectorizer transformer.
3. A BernoulliNB classifier.

The code is as follows:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([('bag-of-words', BagOfWords()), ('vectorizer',
DictVectorizer()), ('naive-bayes', BernoulliNB()) ])
```

We can nearly run our pipeline now, which we will do with `cross_val_score` as we have done many times before. Before we perform the data mining, we will introduce a better evaluation metric than the accuracy metric we used before. As we will see, the use of accuracy is not adequate for datasets when the number of samples in each class is different.

Evaluation using the F1-score

When choosing an evaluation metric, it is always important to consider cases where that evaluation metric is not useful. Accuracy is a good evaluation metric in many cases, as it is easy to understand and simple to compute. However, it can be easily faked. In other words, in many cases, you can create algorithms that have a high accuracy but have a poor utility.

While our dataset of tweets (typically, your results may vary) contains about 50 percent programming-related and 50 percent nonprogramming, many datasets aren't as *balanced* as this.

As an example, an e-mail spam filter may expect to see more than 80 percent of incoming e-mails be spam. A spam filter that simply labels everything as spam is quite useless; however, it will obtain an accuracy of 80 percent!

To get around this problem, we can use other evaluation metrics. One of the most commonly employed is called an **f1-score** (also called f-score, f-measure, or one of many other variations on this term).

The F1-score is defined on a per-class basis and is based on two concepts: the precision and recall. The precision is the percentage of all the samples that were predicted as belonging to a specific class, that were actually from that class. The recall is the percentage of samples in the dataset that are in a class and actually labeled as belonging to that class.

In the case of our application, we could compute the value for both classes (python-programming and not python-programming).

Our precision computation becomes the question: *of all the tweets that were predicted as being relevant, what percentage were actually relevant?*

Likewise, the recall becomes the question: *of all the relevant tweets in the data set, how many were predicted as being relevant?*

After you compute both the precision and recall, the f1-score is the harmonic mean of the precision and recall:

To use the f1-score in scikit-learn methods, simply set the scoring parameter to f1. By default, this will return the f1-score of the class with label 1. Running the code on our dataset, we simply use the following line of code:

```
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(pipeline, tweets, labels, scoring='f1')
# We then print out the average of the scores:
import numpy as np
print("Score: {:.3f}".format(np.mean(scores)))
```

The result is 0.684, which means we can accurately determine if a tweet using Python relates to the programing language nearly 70 percent of the time. This is using a dataset with only 300 tweets in it.

Go back and collect more data and you will find that the results increase! Keep in mind that your dataset may differ, and therefore your results would too.

More data usually means a better accuracy, but it is not guaranteed!

Getting useful features from models

One question you may ask is, what are the best features for determining if a tweet is relevant or not? We can extract this information from our Naive Bayes model and find out which features are the best individually, according to Naive Bayes.

First, we fit a new model. While the `cross_val_score` gives us a score across different folds of cross-validated testing data, it doesn't easily give us the trained models themselves. To do this, we simply fit our pipeline with the tweets, creating a new model. The code is as follows:

```
model = pipeline.fit(tweets, labels)
```

Note that we aren't really evaluating the model here, so we don't need to be as careful with the training/testing split. However, before you put these features into practice, you should evaluate on a separate test split. We skip over that here for the sake of clarity.

A pipeline gives you access to the individual steps through the `named_steps` attribute and the name of the step (we defined these names ourselves when we created the pipeline object itself). For instance, we can get the Naive Bayes model:

```
nb = model.named_steps['naive-bayes']
feature_probabilities = nb.feature_log_prob_
```

From this model, we can extract the probabilities for each word. These are stored as log probabilities, which is simply *log(P(A|f))*, where *f* is a given feature.

The reason these are stored as log probabilities is because the actual values are very low. For instance, the first value is -3.486, which correlates to a probability under 0.03 percent. Logarithm probabilities are used in computation involving small probabilities like this as they stop underflow errors where very small values are just rounded to zeros. Given that all of the probabilities are multiplied together, a single value of 0 will result in the whole answer always being 0! Regardless, the relationship between values is still the same; the higher the value, the more useful that feature is.

We can get the most useful features by sorting the array of logarithm probabilities. We want descending order, so we simply negate the values first. The code is as follows:

```
top_features = np.argsort(-nb.feature_log_prob_[1])[:50]
```

The preceding code will just give us the indices and not the actual feature values. This isn't very useful, so we will map the feature's indices to the actual values. The key is the DictVectorizer step of the pipeline, which created the matrices for us. Luckily this also records the mapping, allowing us to find the feature names that correlate to different columns. We can extract the features from that part of the pipeline:

```
dv = model.named_steps['vectorizer']
```

From here, we can print out the names of the top features by looking them up in the `feature_names_` attribute of DictVectorizer. Enter the following lines into a new cell and run it to print out a list of the top features:

```
for i, feature_index in enumerate(top_features):
    print(i, dv.feature_names_[feature_index],
np.exp(feature_probabilities[1][feature_index]))
```

The first few features include : RT, and even Python. These are likely to be noise (although the use of a colon is not very common outside programming), based on the data we collected. Collecting more data is critical to smoothing out these issues. Looking through the list though, we get a number of more obvious programming features:

```
9 for 0.175
14 ) 0.10625
15 ( 0.10625
22 jobs 0.0625
29 Developer 0.05
```

There are some others too that refer to Python in a work context, and therefore might be referring to the programming language (although freelance snake handlers may also use similar terms, they are less common on Twitter).

That last one is usually in the format: *We're looking for a candidate for this job*.

Looking through these features gives us quite a few benefits. We could train people to recognize these tweets, look for commonalities (which give insight into a topic), or even get rid of features that make no sense. For example, the word RT appears quite high in this list; however, this is a common Twitter phrase for retweet (that is, forwarding on someone else's tweet). An expert could decide to remove this word from the list, making the classifier less prone to the noise we introduced by having a small dataset.

Summary

In this chapter, we looked at text mining—how to extract features from text, how to use those features, and ways of extending those features. In doing this, we looked at putting a tweet in context—was this tweet mentioning python referring to the programming language? We downloaded data from a web-based API, getting tweets from the popular microblogging website Twitter. This gave us a dataset that we labeled using a form we built directly in the Jupyter Notebook.

We also looked at reproducibility of experiments. While Twitter doesn't allow you to send copies of your data to others, it allows you to send the tweet's IDs. Using this, we created code that saved the IDs and recreated most of the original dataset. Not all tweets were returned; some had been deleted in the time since the ID list was created and the dataset was reproduced.

We used a Naive Bayes classifier to perform our text classification. This is built upon the Bayes' theorem that uses data to update the model, unlike the frequentist method that often starts with the model first. This allows the model to incorporate and update new data, and incorporate a prior belief. In addition, the naive part allows to easily compute the frequencies without dealing with complex correlations between features.

The features we extracted were word occurrences—did this word occur in this tweet? This model is called bag-of-words. While this discards information about where a word was used, it still achieves a high accuracy on many datasets. This entire pipeline of using the bag-of-words model with Naive Bayes is quite robust. You will find that it can achieve quite good scores on most text-based tasks. It is a great baseline for you, before trying more advanced models. As another advantage, the Naive Bayes classifier doesn't have any parameters that need to be set (although there are some if you wish to do some tinkering).

To extend the work we did in this chapter, first start by collecting more data. You'll need to manually classify these as well, but you'll find some similarities between tweets that might make it easier. For example, there is a field of study called Locality Sensitive Hashes, that determines whether two tweets are similar. Two similar tweets are likely about the same topic. Another method for extending the research is to consider how you would build a model that incorporates the twitter user's history into the equation - in other words, if the user often tweets about python-as-a-programming-language, then they are more likely to be using python in a future tweet.

In the next chapter, we will look at extracting features from another type of data, graphs, in order to make recommendations on who to follow on social media.

7

Follow Recommendations Using Graph Mining

Graphs can be used to represent a wide range of phenomena. This is particularly true for online social networks, and the **Internet of Things** (**IoT**). Graph mining is big business, with websites such as Facebook running on data analysis experiments performed on graphs.

Social media websites are built upon engagement. Users without active news feeds, or interesting friends to follow, do not engage with sites. In contrast, users with more interesting friends and *followees* engage more, see more ads. This leads to larger revenue streams for the website.

In this chapter, we look at how to define similarity on graphs, and how to use them within a data mining context. Again, this is based on a model of the phenomena. We look at some basic graph concepts, like sub-graphs and connected components. This leads to an investigation of cluster analysis, which we delve more deeply into in `Chapter 10`, *Clustering News Articles.*

The topics covered in this chapter include:

- Clustering data to find patterns
- Loading datasets from previous experiments
- Getting follower information from Twitter
- Creating graphs and networks
- Finding subgraphs for cluster analysis

Loading the dataset

In this chapter, our task is to recommend users on online social networks based on shared connections. Our logic is that if two users have the same friends, they are highly similar and worth recommending to each other. We want our recommendations to be of high value. We can only recommend so many people before it becomes tedious, therefore we need to find recommendations that engage users.

To do this, we use the previous chapter's disambiguation model to find only users talking about *Python as a programming language*. In this chapter, we use the results from one data mining experiment as input into another data mining experiment. Once we have our Python programmers selected, we then use their friendships to find clusters of users that are highly similar to each other. The similarity between two users will be defined by how many friends they have in common. Our intuition will be that the more friends two people have in common, the more likely two people are to be friends (and therefore should be on our social media platform).

We are going to create a small social graph from Twitter using the API we introduced in the previous chapter. The data we are looking for is a subset of users interested in a similar topic (again, the Python programming language) and a list of all of their friends (people they follow). With this data, we will check how similar two users are, based on how many friends they have in common.

There are many other online social networks apart from Twitter. The reason we have chosen Twitter for this experiment is that their API makes it quite easy to get this sort of information. The information is available from other sites, such as Facebook, LinkedIn, and Instagram, as well. However, getting this information is more difficult.

To start collecting data, set up a new Jupyter Notebook and an instance of the `twitter` connection, as we did in the previous chapter. You can reuse the app information from the previous chapter or create a new one:

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token,
access_token_secret, consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization, retry=True)
```

Also, set up the filenames. You will want to use a different folder for this experiment from the one you used in `Chapter 6`, *Social Media Insight Using Naive Bayes*, ensuring you do not override your previous dataset!

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "twitter")
output_filename = os.path.join(data_folder, "python_tweets.json")
```

Next, we will need a list of users. We will do a search for tweets, as we did in the previous chapter, and look for those mentioning the word `python`. First, create two lists for storing the tweet's text and the corresponding users. We will need the user IDs later, so we create a dictionary mapping that now. The code is as follows:

```
original_users = []
tweets = []
user_ids = {}
```

We will now perform a search for the word python, as we did in the previous chapter, and iterate over the search results and only saving Tweets with text (as per the last chapter's requirements):

```
search_results = t.search.tweets(q="python", count=100)['statuses']
for tweet in search_results:
    if 'text' in tweet:
        original_users.append(tweet['user']['screen_name'])
        user_ids[tweet['user']['screen_name']] = tweet['user']['id']
        tweets.append(tweet['text'])
```

Running this code will get about 100 tweets, maybe a little fewer in some cases. Not all of them will be related to the programming language, though. We will address that by using the model we trained in the previous chapter.

Classifying with an existing model

As we learned in the previous chapter, not all tweets that mention the word python are going to be relating to the programming language. To do that, we will use the classifier we used in the previous chapter to get tweets based on the programming language. Our classifier wasn't perfect, but it will result in a better specialization than just doing the search alone.

In this case, we are only interested in users who are tweeting about Python, the programming language. We will use our classifier from the last chapter to determine which tweets are related to the programming language. From there, we will select only those users who were tweeting about the programming language.

To do this part of our broader experiment, we first need to save the model from the previous chapter. Open the Jupyter Notebook we made in the last chapter, the one in which we built and trained the classifier.

If you have closed it, then the Jupyter Notebook won't remember what you did, and you will need to run the cells again. To do this, click on the Cell menu on the Notebook and choose Run All.

After all of the cells have computed, choose the final blank cell. If your Notebook doesn't have a blank cell at the end, choose the last cell, select the Insert menu, and select the **Insert Cell Below** option.

We are going to use the `joblib` library to save our model and load it.

`joblib` is included with the `scikit-learn` package as a built-in external package. No extra installation step needed! This library has tools for saving and loading models, and also for simple parallel processing - which is used in `scikit-learn` quite a lot.

First, import the library and create an output filename for our model (make sure the directories exist, or else they won't be created). I've stored this model in my `Models` directory, but you could choose to store them somewhere else. The code is as follows:

```
from sklearn.externals import joblib
output_filename = os.path.join(os.path.expanduser("~"), "Models",
"twitter", "python_context.pkl")
```

Next, we use the `dump` function in `joblib`, which works much like the similarly named version in the `json` library. We pass the model itself and the output filename:

```
joblib.dump(model, output_filename)
```

Running this code will save our model to the given filename. Next, go back to the new Jupyter Notebook you created in the last subsection and load this model.

You will need to set the model's filename again in this Notebook by copying the following code:

```
model_filename = os.path.join(os.path.expanduser("~"), "Models", "twitter",
"python_context.pkl")
```

Make sure the filename is the one you used just before to save the model. Next, we need to recreate our BagOfWords class, as it was a custom-built class and can't be loaded directly by joblib. Simply copy the entire **BagOfWords** class from the previous chapter's code, including its dependencies:

```
import spacy
from sklearn.base import TransformerMixin

# Create a spaCy parser
nlp = spacy.load('en')


class BagOfWords(TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        results = []
        for document in X:
            row = {}
            for word in list(nlp(document, tag=False, parse=False,
entity=False)):
                if len(word.text.strip()): # Ignore words that are just
whitespace
                    row[word.text] = True
                    results.append(row)
        return results
```

In production, you would need to develop your custom transformers in separate, centralized files, and import them into the Notebook instead. This little hack simplifies the workflow, but feel free to experiment with centralizing important code by creating a library of common functionality.

Loading the model now just requires a call to the `load` function of `joblib`:

```
from sklearn.externals import joblib
context_classifier = joblib.load(model_filename)
```

Our **context_classifier** works exactly like the model object of the notebook we saw in `Chapter 6`, *Social Media Insight Using Naive Bayes*, It is an instance of a **Pipeline**, with the same three steps as before (`BagOfWords`, `DictVectorizer`, and a `BernoulliNB` classifier). Calling the predict function on this model gives us a prediction as to whether our tweets are relevant to the programming language. The code is as follows:

```
y_pred = context_classifier.predict(tweets)
```

The *ith* item in `y_pred` will be 1 if the *ith* tweet is (predicted to be) related to the programming language, or else it will be 0. From here, we can get just the tweets that are relevant and their relevant users:

```
relevant_tweets = [tweets[i] for i in range(len(tweets)) if y_pred[i] == 1]
relevant_users = [original_users[i] for i in range(len(tweets)) if
y_pred[i] == 1]
```

Using my data, this comes up to 46 relevant users. A little lower than our 100 tweets/users from before, but now we have a basis for building our social network. We can always add more data to get more users, but 40+ users will be sufficient to go through this chapter as a first pass. I recommend coming back, adding more data, and running the code again, to see what results you obtain.

Getting follower information from Twitter

With our initial set of users, we now need to get the friends of each of these users. A friend is a person whom the user is following. The API for this is called friends/ids, and it has both good and bad points. The good news is that it returns up to 5,000 friend IDs in a single API call. The bad news is that you can only make 15 calls every 15 minutes, which means it will take you at least 1 minute per user to get all followers—more if they have more than 5,000 friends (which happens more often than you may think).

The code is similar to the code from our previous API usage (obtaining tweets). We will package it as a function, as we will use this code in the next two sections. Our function takes a twitter user's ID value, and returns their friends. While it may be surprising to some, many Twitter users have more than 5,000 friends. Due to this we will need to use Twitter's pagination function, which lets Twitter return multiple pages of data through separate API calls. When you ask Twitter for information, it gives you your information along with a cursor, which is an integer that Twitter uses to track your request. If there is no more information, this cursor is 0; otherwise, you can use the supplied cursor to get the next page of results. Passing this cursor lets twitter continue your query, returning the next set of data to you.

In the function, we keep looping while this cursor is not equal to 0 (as, when it is, there is no more data to collect). We then perform a request for the user's followers and add them to our list. We do this in a try block, as there are possible errors that can happen that we can handle. The follower's IDs are stored in the ids key of the results dictionary. After obtaining that information, we update the cursor. It will be used in the next iteration of the loop. Finally, we check if we have more than 10,000 friends. If so, we break out of the loop. The code is as follows:

```
import time

def get_friends(t, user_id):
    friends = []
    cursor = -1
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id, cursor=cursor,
count=5000)
            friends.extend([friend for friend in results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
        except TypeError as e:
            if results is None:
                print("You probably reached your API limit, waiting for 5
minutes")
                sys.stdout.flush()
                time.sleep(5*60) # 5 minute wait
            else:
                # Some other error happened, so raise the error as normal
                raise e
        except twitter.TwitterHTTPError as e:
            print(e)
            break
        finally:
            # Break regardless -- this stops us going over our API limit
            time.sleep(60)
```

It is worth inserting a warning here. We are dealing with data from the Internet, which means weird things can and do happen regularly. A problem I ran into when developing this code was that some users have many, many, many thousands of friends. As a fix for this issue, we will put a failsafe here, exiting the function if we reach more than 10,000 users. If you want to collect the full dataset, you can remove these lines, but beware that it may get stuck on a particular user for a very long time.

Much of the above function is error handling, as quite a lot can go wrong when dealing with external APIs!

The most likely error that can happen is if we accidentally reach our API limit (while we have a sleep to stop that, it can occur if you stop and run your code before this sleep finishes). In this case, results is `None` and our code will fail with a `TypeError`. In this case, we wait for 5 minutes and try again, hoping that we have reached our next 15-minute window. There may be another `TypeError` that occurs at this time. If one of them does, we raise it and will need to handle it separately.

The second error that can happen occurs at Twitter's end, such as asking for a user that doesn't exist or some other data-based error, resulting in a `TwitterHTTPError` (which is a similar concept to a HTTP 404 error). In this case, don't try this user anymore, and just return any followers we did get (which, in this case, is likely to be 0).

Finally, Twitter only lets us ask for follower information 15 times every 15 minutes, so we will wait for 1 minute before continuing. We do this in a **finally** block so that it happens even if an error occurs.

Building the network

Now we are going to build our network of users, where users are linked if the two users follow each other. The aim of building this network is to give us a data structure we can use to segment our list of users into groups. From these groups, we can then recommend people in the same group to each other. Starting with our original users, we will get the friends for each of them and store them in a dictionary. Using this concept we can grow the graph outwards from an initial set of users.

Starting with our original users, we will get the friends for each of them and store them in a dictionary (after obtaining the user's ID from our **user_id** dictionary):

```
friends = {}
for screen_name in relevant_users:
    user_id = user_ids[screen_name]
    friends[user_id] = get_friends(t, user_id)
```

Next, we are going to remove any user who doesn't have any friends. For these users, we can't really make a recommendation in this way. Instead, we might have to look at their content or people who follow them. We will leave that out of the scope of this chapter, though, so let's just remove these users. The code is as follows:

```
friends = {user_id:friends[user_id]
           for user_id in friends
           if len(friends[user_id]) > 0}
```

We now have between 30 and 50 users, depending on your initial search results. We are now going to increase that amount to 150. The following code will take quite a long time to run—given the limits on the API, we can only get the friends for a user once every minute. Simple math will tell us that 150 users will take 150 minutes, which is at least 2 hours and 30 minutes. Given the time we are going to be spending on getting this data, it pays to ensure we get only good users.

What makes a good user, though? Given that we will be looking to make recommendations based on shared connections, we will search for users based on shared connections. We will get the friends of our existing users, starting with those users who are better connected to our existing users. To do that, we maintain a count of all the times a user is in one of our friend's lists. It is worth considering the goals of the application when considering your sampling strategy. For this purpose, getting lots of similar users enables the recommendations to be more regularly applicable.

To do this, we simply iterate over all the friend lists we have and then count each time a friend occurs.

```
from collections import defaultdict
def count_friends(friends):
    friend_count = defaultdict(int)
    for friend_list in friends.values():
        for friend in friend_list:
            friend_count[friend] += 1
    return friend_count
```

Computing our current friend count, we can then get the most connected (that is, most friends from our existing list) person from our sample. The code is as follows:

```
friend_count = count_friends(friends)
from operator import itemgetter
best_friends = sorted(friend_count, key=friend_count.get, reverse=True)
```

From here, we set up a loop that continues until we have the friends of 150 users. We then iterate over all of our best friends (which happens in order of the number of people who have them as friends) until we find a user we have not yet checked. We then get the friends of that user and update the `friends` counts. Finally, we work out who is the most connected user who we haven't already got in our list:

```
while len(friends) < 150:
    for user_id, count in best_friends:
        if user_id in friends:
            # Already have this user, move to next one
            continue
        friends[user_id] = get_friends(t, user_id)
        for friend in friends[user_id]:
            friend_count[friend] += 1
        best_friends = sorted(friend_count.items(), key=itemgetter(1),
reverse=True)
        break
```

The codes will then loop and continue until we reach 150 users.

You may want to set these values lower, such as 40 or 50 users (or even just skip this bit of code temporarily). Then, complete the chapter's code and get a feel for how the results work. After that, reset the number of users in this loop to 150, leave the code to run for a few hours, and then come back and rerun the later code.

Given that collecting that data probably took nearly 3 hours, it would be a good idea to save it in case we have to turn our computer off. Using the `json` library, we can easily save our friends dictionary to a file:

```
import json
friends_filename = os.path.join(data_folder, "python_friends.json")
with open(friends_filename, 'w') as outf:
    json.dump(friends, outf)
```

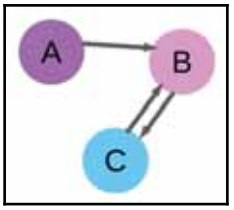
If you need to load the file, use the `json.load` function:

```
with open(friends_filename) as inf:
    friends = json.load(inf)
```

Creating a graph

At this point in our experiment, we have a list of users and their friends. This gives us a graph where some users are friends of other users (although not necessarily the other way around).

A **graph** is a set of nodes and edges. Nodes are usually objects of interest - in this case, they are our users. The edges in this initial graph indicate that user A is a friend of user B. We call this a **directed graph**, as the order of the nodes matters. Just because user A is a friend of user B, that doesn't imply that user B is a friend of user A. The example network below shows this, along with a user C who is friends of user B, and is friended in turn by user B as well:



In python, one of the best libraries for working with graphs, including creating, visualising and computing, is called **NetworkX**.

Once again, you can use Anaconda to install NetworkX: `conda install networkx`

First, we create a directed graph using NetworkX. By convention, when importing NetworkX, we use the abbreviation nx (although this isn't necessary). The code is as follows:

```
import networkx as nx
G = nx.DiGraph()
```

We will only visualize our key users, not all of the friends (as there are many thousands of these and it is hard to visualize). We get our main users and then add them to our graph as nodes:

```
main_users = friends.keys()
G.add_nodes_from(main_users)
```

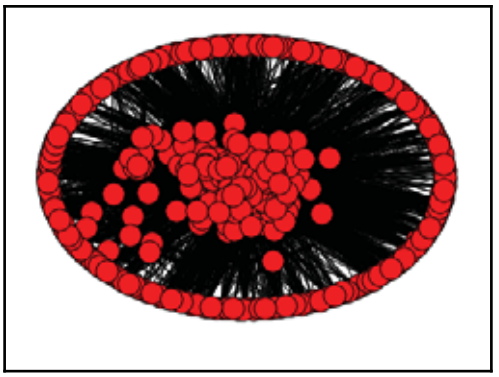
Next we set up the edges. We create an edge from a user to another user if the second user is a friend of the first user. To do this, we iterate through all of the friends of a given user. We ensure that the friend is one of our main users (as we currently aren't interested in visualizing the other users), and add the edge if they are.

```
for user_id in friends:
    for friend in friends[user_id]:
        if str(friend) in main_users:
            G.add_edge(user_id, friend)
```

We can now visualize the network using NetworkX's draw function, which uses matplotlib. To get the image in our notebook, we use the inline function on matplotlib and then call the draw function. The code is as follows:

```
%matplotlib inline
nx.draw(G)
```

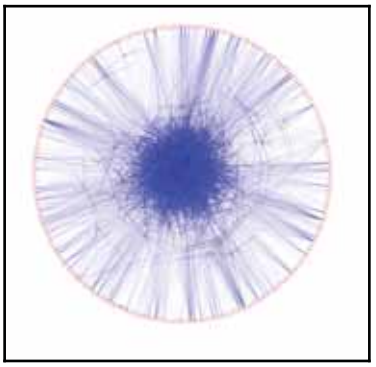
The results are a bit hard to make sense of; they show just the ring of nodes, and its hard to work anything specific out about the dataset. Not a good image at all:



We can make the graph a bit better by using **pyplot** to handle the creation of the figure, which is used by NetworkX to do graph drawing. Import `pyplot`, create a larger figure, and then call NetworkX's `draw` function to increase the size of the image:

```
from matplotlib import pyplot as plt
plt.figure(3,figsize=(20,20))
nx.draw(G, alpha=0.1, edge_color='b')
```

By making the graph bigger and adding transparency, an outline of how the graph appears can now be seen:



In my graph, there was a major group of users all highly connected to each other, and most other users didn't have many connections at all. As you can see, it is very well connected in the center!

This is actually a property of our method of choosing new users—we choose those who are already well linked in our graph, so it is likely they will just make this group larger. For social networks, generally the number of connections a user has follows a power law. A small percentage of users have many connections, and others have only a few. The shape of the graph is often described as having a *long tail*.

By zooming into parts of the graph you can start seeing structure. Visualizing and analyzing graphs like this is hard - we will see some tools for making this process easier in the next section.

Creating a similarity graph

The final step to this experiment is to recommend users, based on how many friends they share. As mentioned previously, our logic is that, if two users have the same friends, they are highly similar. We could recommend one user to the other on this basis.

We are therefore going to take our existing graph (which has edges relating to friendship) and create a new graph from its information. The nodes are still users, but the edges are going to be **weighted edges**. A weighted edge is simply an edge with a weight property. The logic is that a higher weight indicates more similarity between the two nodes than a lower weight. This is context-dependent. If the weights represent distance, then the lower weights indicate more similarity.

For our application, the weight will be the similarity of the two users connected by that edge (based on the number of friends they share). This graph also has the property that it is not directed. This is due to our similarity computation, where the similarity of user A to user B is the same as the similarity of user B to user A.

Other similarity measurements are directed. An example is ratio of similar users, which is the number of friends in common divided by the user's total number of friends. In this case, you would need a directed graph.

There are many ways to compute the similarity between two lists like this. For example, we could compute the number of friends the two have in common. However, this measure is always going to be higher for people with more friends. Instead, we can normalize it by dividing by the total number of distinct friends the two have. This is called the **Jaccard Similarity**.

The Jaccard Similarity, always between 0 and 1, represents the percentage overlap of the two. As we saw in `Chapter 2`, *Classifying with scikit-learn Estimators*, normalization is an important part of data mining exercises and generally a good thing to do. There are fringe cases where you wouldn't normalize data, but by default normalize first.

To compute the Jaccard similarity, we divide the intersection of the two sets of followers by the union of the two. These are set operations and we have lists, so we will need to convert the friends lists to **sets** first. The code is as follows:

```
friends = {user: set(friends[user]) for user in friends}
```

We then create a function that computes the similarity of two sets of friends lists. The code is as follows:

```
def compute_similarity(friends1, friends2):
    return len(friends1 & friends2) / (len(friends1 | friends2)  + 1e-6)
```

We add **1e-6** (or 0.000001) to the similarity above to ensure we never get a division by zero error, in cases where neither user has any friends. It is small enough to not really affect our results, but big enough to be more than zero.

From here, we can create our weighted graph of the similarity between users. We will use this quite a lot in the rest of the chapter, so we will create a function to perform this action. Let's take a look at the threshold parameter:

```
def create_graph(followers, threshold=0):
    G = nx.Graph()
    for user1 in friends.keys():
```

```
        for user2 in friends.keys():
            if user1 == user2:
                continue
            weight = compute_similarity(friends[user1], friends[user2])
            if weight >= threshold:
                G.add_node(user1)
                G.add_node(user2)
                G.add_edge(user1, user2, weight=weight)
    return G
```

We can now create a graph by calling this function. We start with no threshold, which means all links are created. The code is as follows:

```
G = create_graph(friends)
```

The result is a very strongly connected graph—all nodes have edges, although many of those will have a weight of 0. We will see the weight of the edges by drawing the graph with line widths relative to the weight of the edge—thicker lines indicate higher weights.

Due to the number of nodes, it makes sense to make the figure larger to get a clearer sense of the connections:

```
plt.figure(figsize=(10,10))
```

We are going to draw the edges with a weight, so we need to draw the nodes first. NetworkX uses **layouts** to determine where to put the nodes and edges, based on certain criteria. Visualizing networks is a very difficult problem, especially as the number of nodes grows. Various techniques exist for visualizing networks, but the degree to which they work depends heavily on your dataset, personal preferences, and the aim of the visualization. I found that the **spring_layout** worked quite well, but other options such as **circular_layout** (which is a good default if nothing else works), **random_layout**, **shell_layout**, and **spectral_layout** also exist and have uses where the others may fail.

Visit `http://networkx.lanl.gov/reference/drawing.html` for more details on layouts in NetworkX. Although it adds some complexity, the `draw_graphviz` option works quite well and is worth investigating for better visualizations. It is well worth considering in real-world uses.

Let's use `spring_layout` for visualization:

```
pos = nx.spring_layout(G)
```

Using our `pos` layout, we can then position the nodes:

```
nx.draw_networkx_nodes(G, pos)
```

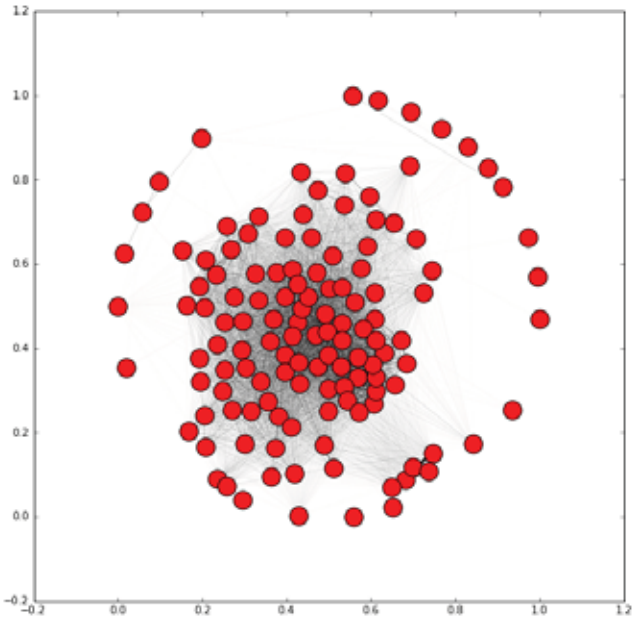
Next, we draw the edges. To get the weights, we iterate over the edges in the graph (in a specific order) and collect the weights:

```
edgewidth = [ d['weight'] for (u,v,d) in G.edges(data=True)]
```

We then draw the edges:

```
nx.draw_networkx_edges(G, pos, width=edgewidth)
```

The result will depend on your data, but it will typically show a graph with a large set of nodes connected quite strongly and a few nodes poorly connected to the rest of the network.



The difference in this graph compared to the previous graph is that the edges determine the similarity between the nodes based on our similarity metric and not on whether one is a friend of another (although there are similarities between the two!). We can now start extracting information from this graph in order to make our recommendations.

Finding subgraphs

From our similarity function, we could simply rank the results for each user, returning the most similar user as a recommendation - as we did with our product recommendations. This works, and is indeed one way to perform this type of analysis.

Instead, we might want to find clusters of users that are all similar to each other. We could advise these users to start a group, create advertising targeting this segment, or even just use those clusters to do the recommendations themselves. Finding these clusters of similar users is a task called **cluster analysis**.

Cluster analysis is a difficult task, with complications that classification tasks do not typically have. For example, evaluating classification results is relatively easy - we compare our results to the ground truth (from our training set) and see what percentage we got right. With cluster analysis, though, there isn't typically a ground truth. Evaluation usually comes down to seeing if the clusters make sense, based on some preconceived notion we have of what the cluster should look like.

Another complication with cluster analysis is that the model can't be trained against the expected result to learn—it has to use some approximation based on a mathematical model of a cluster, not what the user is hoping to achieve from the analysis.

Due to these issues, cluster analysis is more of an exploratory tool, rather than a prediction tool. Some research and applications use clustering for analysis, but its usefulness as a predictive model is dependent on an analyst selecting parameters and finding graphs that *look right*, rather than a specific evaluation metric.

Connected components

One of the simplest methods for clustering is to find the **connected components** in a graph. A connected component is a set of nodes in a graph that are connected via edges. Not all nodes need to be connected to each other to be a connected component. However, for two nodes to be in the same connected component, there needs to be a way to *travel* from one node to another in that connected component by moving along edges.

Connected components do not consider edge weights when being computed; they only check for the presence of an edge. For that reason, the code that follows will remove any edge with a low weight.

NetworkX has a function for computing connected components that we can call on our graph. First, we create a new graph using our `create_graph` function, but this time we pass a threshold of 0.1 to get only those edges that have a weight of at least 0.1, indicative of 10% of followers in common between the two node users:

```
G = create_graph(friends, 0.1)
```

We then use NetworkX to find the connected components in the graph:

```
sub_graphs = nx.connected_component_subgraphs(G)
```

To get a sense of the sizes of the graph, we can iterate over the groups and print out some basic information:

```
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

The results will tell you how big each of the connected components is. My results had one large subgraph of 62 users and lots of little ones with a dozen or fewer users.

We can alter the **threshold** to alter the connected components. This is because a higher threshold has fewer edges connecting nodes, and therefore will have smaller connected components and more of them. We can see this by running the preceding code with a higher threshold:

```
G = create_graph(friends, 0.25)
sub_graphs = nx.connected_component_subgraphs(G)
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

The preceding code gives us much smaller subgraphs and more of them. My largest cluster was broken into at least three parts and none of the clusters had more than 10 users. An example cluster is shown in the following figure, and the connections within this cluster are also shown. Note that, as it is a connected component, there were no edges from nodes in this component to other nodes in the graph (at least, with the threshold set at 0.25).

We can draw the entire graph, showing each connected component in a different color. As these connected components are not connected to each other, it actually makes little sense to plot these on a single graph. This is because the positioning of the nodes and components is arbitrary, and it can confuse the visualization. Instead, we can plot each separately on a separate subfigure.

In a new cell, obtain the connected components and also the count of the connected components:

```
sub_graphs = nx.connected_component_subgraphs(G)
n_subgraphs = nx.number_connected_components(G)
```

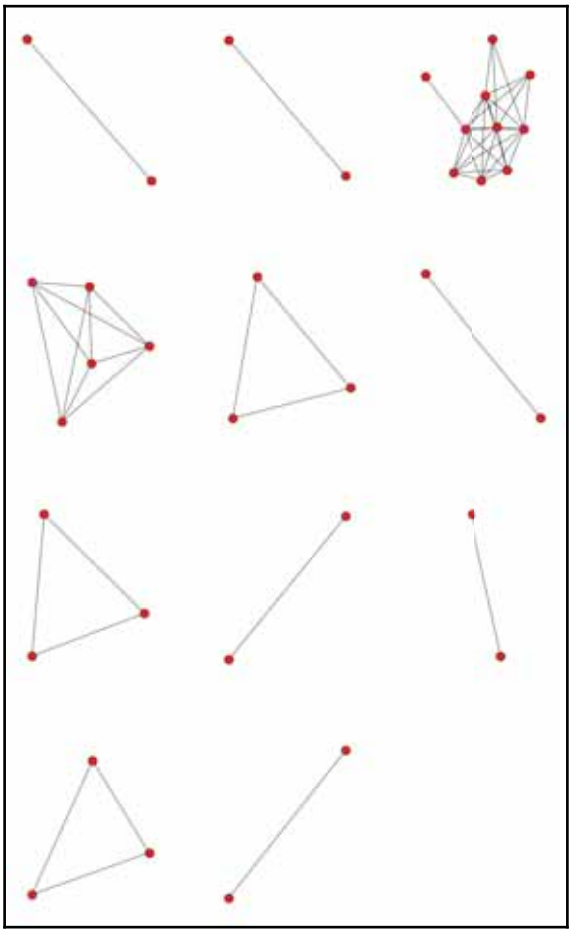
`sub_graphs` is a generator, not a list of the connected components. For this reason, use `nx.number_connected_components` to find out how many connected components there are; don't use `len`, as it doesn't work due to the way that NetworkX stores this information. This is why we need to recompute the connected components here.

Create a new pyplot figure and give enough room to show all of our connected components. For this reason, we allow the graph to increase in size with the number of connected components.

Next, iterate over each connected component and add a subplot for each. The parameters to add_subplot are the number of rows of subplots, the number of columns, and the index of the subplot we are interested in. My visualization uses three columns, but you can try other values instead of three (just remember to change both values):

```
fig = plt.figure(figsize=(20, (n_subgraphs * 3)))
for i, sub_graph in enumerate(sub_graphs):
    ax = fig.add_subplot(int(n_subgraphs / 3) + 1, 3, i + 1)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, sub_graph.nodes(), ax=ax, node_size=500)
    nx.draw_networkx_edges(G, pos, sub_graph.edges(), ax=ax)
```

The results visualize each connected component, giving us a sense of the number of nodes in each and also how connected they are.



If you are not seeing anything on your graphs, try rerunning the line:
`sub_graphs = nx.connected_component_subgraphs(G)`
The `sub_graphs` object is a generator and is "consumed" after being used.

Optimizing criteria

Our algorithm for finding these connected components relies on the **threshold** parameter, which dictates whether edges are added to the graph or not. In turn, this directly dictates how many connected components we discover and how big they are. From here, we probably want to settle on some notion of which is the *best* threshold to use. This is a very subjective problem, and there is no definitive answer. This is a major problem with any cluster analysis task.

We can, however, determine what we think a good solution should look like and define a metric based on that idea. As a general rule, we usually want a solution where:

- Samples in the same cluster (connected components) are highly *similar* to each other
- Samples in different clusters are highly *dissimilar* to each other

The **Silhouette Coefficient** is a metric that quantifies these points. Given a single sample, we define the Silhouette Coefficient as follows:

$$s = \frac{b - a}{\max(a, b)}$$

Where a is the **intra-cluster distance** or the average distance to the other samples in the sample's cluster, and b is the **inter-cluster distance** or the average distance to the other samples in the *next-nearest* cluster.

To compute the overall Silhouette Coefficient, we take the mean of the Silhouette Coefficients for each sample. A clustering that provides a Silhouette Coefficient close to the maximum of 1 has clusters that have samples all similar to each other, and these clusters are very spread apart. Values near 0 indicate that the clusters all overlap and there is little distinction between clusters. Values close to the minimum of -1 indicate that samples are probably in the wrong cluster, that is, they would be better off in other clusters.

Using this metric, we want to find a solution (that is, a value for the threshold) that maximizes the Silhouette Coefficient by altering the threshold parameter. To do that, we create a function that takes the threshold as a parameter and computes the Silhouette Coefficient.

We then pass this into the **optimize** module of SciPy, which contains the `minimize` function that is used to find the minimum value of a function by altering one of the parameters. While we are interested in maximizing the Silhouette Coefficient, SciPy doesn't have a maximize function. Instead, we minimize the inverse of the Silhouette (which is basically the same thing).

The scikit-learn library has a function for computing the Silhouette Coefficient, `sklearn.metrics.silhouette_score`; however, it doesn't fix the function format that is required by the SciPy minimize function. The minimize function requires the variable parameter to be first (in our case, the threshold value), and any arguments to be after it. In our case, we need to pass the friends dictionary as an argument in order to compute the graph.

The Silhouette Coefficient is not defined unless there are at least two nodes (in order for distance to be computed at all). In this case, we define the problem scope as invalid. There are a few ways to handle this, but the easiest is to return a very poor score. In our case, the minimum value that the Silhouette Coefficient can take is -1, and we will return -99 to indicate an invalid problem. Any valid solution will score higher than this.

The function below incorporates all of these issues giving us a function that takes a threshold value and a friends list, and computes the Silhouette Coefficient. It does this by building a matrix from the graph using NetworkX's `to_scipy_sparse_matrix` function.

```
import numpy as np
from sklearn.metrics import silhouette_score

def compute_silhouette(threshold, friends):
    G = create_graph(friends, threshold=threshold)
    if len(G.nodes()) < 2:
        return -99
    sub_graphs = nx.connected_component_subgraphs(G)

    if not (2 <= nx.number_connected_components() < len(G.nodes()) - 1):
        return -99

    label_dict = {}
    for i, sub_graph in enumerate(sub_graphs):
        for node in sub_graph.nodes():
            label_dict[node] = i
```

```
    labels = np.array([label_dict[node] for node in G.nodes()])
    X = nx.to_scipy_sparse_matrix(G).todense()
    X = 1 - X
    return silhouette_score(X, labels, metric='precomputed')
```

For evaluating sparse datasets, I recommend that you look into V-Measure or Adjusted Mutual Information. These are both implemented in scikit-learn, but they have very different parameters for performing their evaluation.

The Silhouette Coefficient implementation in scikit-learn, at the time of writing, doesn't support sparse matrices. For this reason, we need to call the `todense` function. Typically, this is a bad idea--sparse matrices are usually used because the data typically shouldn't be in a dense format. In this case, it will be fine because our dataset is relatively small; however, don't try this for larger datasets.

We have two forms of inversion happening here. The first is taking the inverse of the similarity to compute a distance function; this is needed, as the Silhouette Coefficient only accepts distances. The second is the inverting of the Silhouette Coefficient score so that we can minimize with SciPy's optimize module.

Finally, we create the function that we will minimize. This function is the inverse of the `compute_silhouette` function, because we want lower scores to be better. We could do this in our `compute_silhouette` function--I've separated them here to clarify the different steps involved.

```
def inverted_silhouette(threshold, friends):
    return -compute_silhouette(threshold, friends)
```

This function creates a new function from an original function. When the new function is called, all of the same arguments and keywords are passed onto the original function and the return value is returned, except that this returned value is negated before it is returned.

Now we can do our actual optimization. We call the minimize function on the inverted `compute_silhouette` function we defined:

```
from scipy.optimize import minimize
result = minimize(inverted_silhouette, 0.1, args=(friends,))
```

This function will take quite a while to run. Our graph creation function isn't that fast, nor is the function that computes the Silhouette Coefficient. Decreasing the `maxiter` parameter's value will result in fewer iterations being performed, but we run the risk of finding a suboptimal solution.

Running this function, I got a threshold of 0.135 that returns 10 components. The score returned by the minimize function was -0.192. However, we must remember that we negated this value. This means our score was actually 0.192. The value is positive, which indicates that the clusters tend to be better separated than not (a good thing). We could run other models and check whether it results in a better score, which means that the clusters are better separated.

We could use this result to recommend users—if a user is in a specific connected component, then we can recommend other users in that same component. This recommendation follows our use of the Jaccard Similarity to find good connections between users, our use of connected components to split them up into clusters, and our use of the optimization technique to find the best model in this setting.

However, a large number of users may not be connected at all, so we will use a different algorithm to find clusters for them. We will see other methods for cluster analysis in `Chapter 10`, *Clustering News Articles*.

Summary

In this chapter, we looked at graphs from social networks and how to do cluster analysis on them. We also looked at saving and loading models from scikit-learn by using the classification model we created in `Chapter 6`, *Social Media Insight Using Naive Bayes.*

We created a graph of friends from the social network Twitter. We then examined how similar two users were, based on their friends. Users with more friends in common were considered more similar, although we normalize this by considering the overall number of friends they have. This is a commonly used way to infer knowledge (such as age or general topic of discussion) based on similar users. We can use this logic for recommending users to others—if they follow user X and user Y is similar to user X, they will probably like user Y. This is, in many ways, similar to our transaction-led similarity of previous chapters.

The aim of this analysis was to recommend users, and our use of cluster analysis allowed us to find clusters of similar users. To do this, we found connected components on a weighted graph we created based on this similarity metric. We used the NetworkX package for creating graphs, using our graphs, and finding these connected components.

We then used the Silhouette Coefficient, which is a metric that evaluates how good a clustering solution is. Higher scores indicate a better clustering, according to the concepts of intracluster and intercluster distance. SciPy's optimize module was used to find the solution that maximizes this value.

In this chapter, we saw a few opposites in action. Similarity is a measure between two objects, where higher values indicate more similarity between those objects. In contrast, distance is a measure where lower values indicate more similarity. Another contrast we saw was a loss function, where lower scores are considered better (that is, we lost less). Its opposite is the score function, where higher scores are considered better.

To extend the work in this chapter, examine the V-measure and Adjusted Mutual Information scores in scikit-learn. These replace the Silhouette Coefficient used in this chapter. Are the clusters that result from maximizing these metrics better than the Silhouette Coefficient's clusters? Further, how can you tell? Often, the problem with cluster analysis is that you cannot objectively tell and may use human intervention to choose the best option.

In the next chapter, we will see how to extract features from another new type of data--images. We will discuss how to use neural networks to identify numbers in images and develop a program to automatically beat CAPTCHA images.

8

Beating CAPTCHAs with Neural Networks

Images pose interesting and difficult challenges for data miners. Until recently, only small amounts of progress were made with analyzing images for extracting information. However recently, such as with the progress made on self-driving cars, significant advances have been made in a very short time-frame. The latest research is providing algorithms that can understand images for commercial surveillance, self-driving vehicles, and person identification.

There is lots of raw data in an image, and the standard method for encoding images - pixels - isn't that informative by itself. Images and photos can be blurry, too close to the targets, too dark, too light, scaled, cropped, skewed, or any other of a variety of problems that cause havoc for a computer system trying to extract useful information. Neural networks can combine these lower level features into higher level patterns that are more able to generalize and deal with these issues.

In this chapter, we look at extracting text data from images by using neural networks for predicting each letter in the CAPTCHA. CAPTCHAs are images designed to be easy for humans to solve and hard for a computer to solve, as per the acronym: **Completely Automated Public Turing test to tell Computers and Humans Apart**. Many websites use them for registration and commenting systems to stop automated programs flooding their site with fake accounts and spam comments.

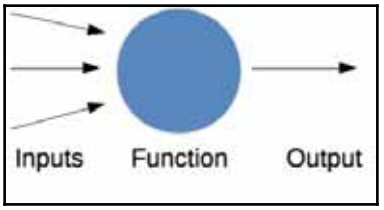
These tests help stop programs (bots) using websites, such as a bot intent on automatically signing up new people to a website. We play the part of such a spammer, trying to get around a CAPTCHA-protected system for posting messages to an online forum. The website is protected by a CAPTCHA, meaning we can't post unless we pass the test.

The topics covered in this chapter include:

- Neural networks
- Creating our own dataset of CAPTCHAs and letters
- The scikit-image library for working with image data
- Extracting basic features from images
- Using neural networks for larger-scale classification tasks
- Improving performance using postprocessing
- Artificial neural networks

Artificial neural networks

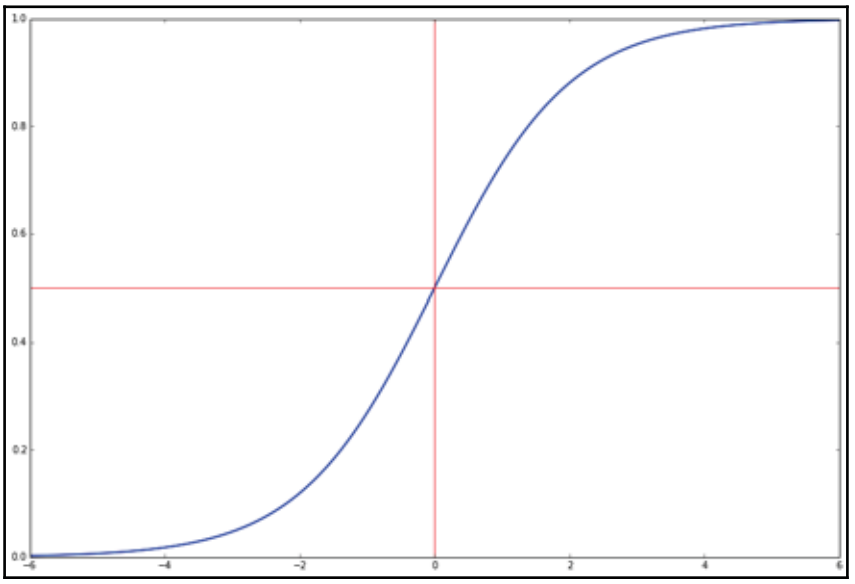
Neural networks are a class of algorithm that was originally designed based on the way that human brains work. However, modern advances are generally based on mathematics rather than biological insights. A neural network is a collection of neurons that are connected together. Each neuron is a simple function of its inputs, which are combined using some function to generate an output:



The functions that define a neuron's processing can be any standard function, such as a linear combination of the inputs, and is called the **activation function**. For the commonly used learning algorithms to work, we need the activation function to be *derivable* and *smooth*. A frequently used activation function is the **logistic function**, which is defined by the following equation (*k* is often simply 1, *x* is the inputs into the neuron, and L is normally 1, that is, the maximum value of the function):

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}}$$

The value of this graph, from -6 to +6, is shown below. The red lines indicate that the value is 0.5 when *x* is zero, but the function quickly climbs to 1.0 as x increases, and quickly drops to -1.0 when x decreases.

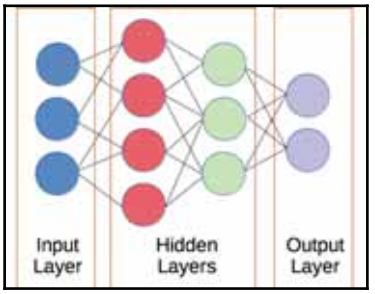


Each individual neuron receives its inputs and then computes the output based on these values. Neural networks can be considered as a collection of these neurons connected together, and they can be very powerful for data mining applications. The combinations of these neurons, how they fit together, and how they combine to learn a model are one of the most powerful concepts in machine learning.

An introduction to neural networks

For data mining applications, the arrangement of neurons is usually in **layers**. The first layer is called the **input layer** and takes its input from samples in the data. The outputs of each of these neurons are computed and then passed along to the neurons in the next layer. This is called a **feed-forward neural network**. We will refer to these simply as **neural networks** for this chapter, as they are the most common type used and the only type used in this chapter. There are other types of neural networks too that are used for different applications. We will see another type of network in `Chapter 11`, *Object Detection in Images Using Deep Neural Networks*.

The outputs of one layer become the inputs of the next layer, continuing until we reach the final layer: the **output layer**. These outputs represent the predictions of the neural network as the classification. Any layer of neurons between the input layer and the output layer is referred to as a **hidden layer**, as they learn a representation of the data not intuitively interpretable by humans. Most neural networks have at least three layers, although most modern applications use networks with many more layers than that.



Typically, we consider fully connected layers. The outputs of each neuron in a layer go to all neurons in the next layer. While we do define a fully connected network, many of the weights will be set to zero during the training process, effectively removing these links. Additionally, many of these weights might retain very small values, even after training.

In addition to being one of the conceptually simpler forms for neural networks, fully connected neural networks are also simpler and more efficient to program than other connection patterns.

See `Chapter 11`, *Object Detection in images using Deep Neural Networks*, for an investigation into different types of neural networks, including layers built specifically for image processing.

As the function of the neurons is normally the logistic function, and the neurons are fully connected to the next layer, the parameters for building and training a neural network must be other factors.

- The first factor for neural networks is in the building phase: the size and shape of the neural network. This includes how many layers the neural network has and how many neurons it has in each hidden layer (the size of the input and output layers is usually dictated by the dataset).

- The second parameter for neural networks is determined in the training phase: the weight of the connections between neurons. When one neuron connects to another, this connection has an associated weight that is multiplied by the signal (the output of the first neuron). If the connection has a weight of 0.8, the neuron is activated, and it outputs a value of 1, the resulting input to the next neuron is 0.8. If the first neuron is not activated and has a value of 0, this stays at 0.

The combination of an appropriately sized network and well-trained weights determines how accurate the neural network can be when making classifications. The word *appropriately* in the previous sentence also doesn't necessarily mean bigger, as neural networks that are too large can take a long time to train and can more easily over-fit the training data.

Weights can be set randomly to start with but are then updated during the training phase. Setting weights to zero is normally not a good idea, as all neurons in the network act similarly to begin with! Having randomly set weights gives each neuron a different *role* in the learning process that can be improved with training.

A neural network in this configuration is a classifier that can then be used to predict the target of a data sample based on the inputs, much like the classification algorithms we have used in previous chapters. But first, we need a dataset to train and test with.

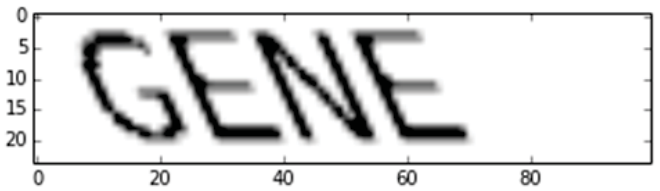
Neural networks are, by a margin, the biggest area of advancement in data mining in recent years. This might make you think: *Why bother learning any other type of classification algorithm?* While neural networks are state of the art in pretty much every domain (at least, right now), the reason to learn other classifiers is that neural networks often require larger amounts of data to work well, and they take a long time to learn. If you don't have **big data**, you will probably get better results from another algorithm.

Creating the dataset

In this chapter, to spice up things a little, let us take on the role of the bad guy. We want to create a program that can beat CAPTCHAs, allowing our comment spam program to advertise on someone's website. It should be noted that our CAPTCHAs will be a little easier than those used on the web today and that spamming isn't a very nice thing to do.

We play the bad guy today, but please *don't* use this against real world sites. One reason to "play the bad guy" is to help improve the security of our website, by looking for issues with it.

Our experiment will simplify a CAPTCHA to be individual English words of four letters only, as shown in the following image:



Our goal will be to create a program that can recover the word from images like this. To do this, we will use four steps:

1. Break the image into individual letters.
2. Classify each individual letter.
3. Recombine the letters to form a word.
4. Rank words with a dictionary to try to fix errors.

Our CAPTCHA-busting algorithm will make the following assumptions. First, the word will be a whole and valid four-character English word (in fact, we use the same dictionary for creating and busting CAPTCHAs). Second, the word will only contain uppercase letters. No symbols, numbers, or spaces will be used.

We are going to make the problem slightly harder than simply identifying letters, by performing a shear transform to the text, along with varying rates of shearing and scaling.

Drawing basic CAPTCHAs

Before we can start classifying CAPTCHAs, we first need a dataset to learn from. In this section, we will be generating our own data to perform the data mining on.

In more real-world applications, you'll be wanting to use an existing CAPTCHA service to generate the data, but for our purposes in this chapter, our own data will be sufficient. One of the issues that can arise is that we code in our assumptions around how the data works when we create the dataset ourselves, and then carry those same assumptions over to our data mining training.

Our goal here is to draw an image with a word on it, along with a shear transform. We are going to use the PIL library to draw our CAPTCHAs and the `scikit-image` library to perform the shear transform. The `scikit-image` library can read images in a NumPy array format that PIL can export to, allowing us to use both libraries.

Both PIL and scikit-image can be installed via Anaconda. However, I recommend getting PIL through its replacement called **pillow**:
conda install pillow scikit-image

First, we import the necessary libraries and modules. We import NumPy and the Image drawing functions as follows:

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf
```

Then we create our base function for generating CAPTCHAs. This function takes a word and a shear value (which is normally between 0 and 0.5) to return an image in a NumPy array format. We allow the user to set the size of the resulting image, as we will use this function for single-letter training samples as well:

```
def create_captcha(text, shear=0, size=(100, 30), scale=1):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(r"bretan/Coval-Black.otf", 22)
    draw.text((0, 0), text, fill=1, font=font)
    image = np.array(im)
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    image = image / image.max()
    # Apply scale
    shape = image.shape
    shapex, shapey = (int(shape[0] * scale), int(shape[1] * scale))
    image = tf.resize(image, (shapex, shapey))
    return image
```

In this function, we create a new image using L for the format, which means black-and-white pixels only, and create an instance of the `ImageDraw` class. This allows us to draw on this image using PIL. We then load the font, draw the text, and perform a `scikit-image` shear transform on it.

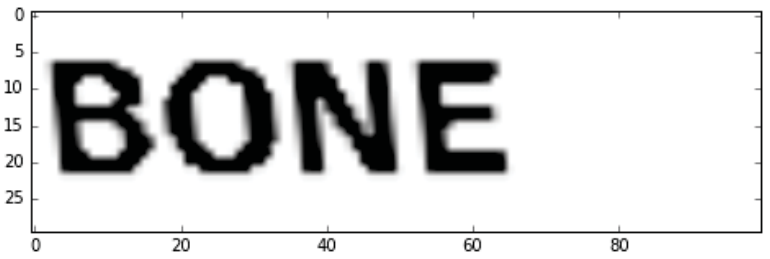
You can get the Coval font I used from the Open Font Library at: `http://openfontlibrary.org/en/font/bretan`
Download the `.zip` file and extract the `Coval-Black.otf` file into the same directory as your Notebook.

From here, we can now generate images quite easily and use `pyplot` to display them. First, we use our inline display for the matplotlib graphs and import `pyplot`. The code is as follows:

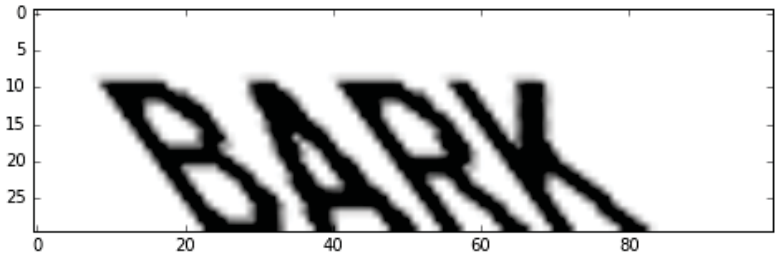
```
%matplotlib inline
from matplotlib import pyplot as plt
image = create_captcha("GENE", shear=0.5, scale=0.6)
plt.imshow(image, cmap='Greys')
```

The result is the image shown at the start of this section: our CAPTCHA. Here are some other examples with different shear and scale values:

```
image = create_captcha("SEND", shear=0.1, scale=1.0)
plt.imshow(image, cmap='Greys')
```

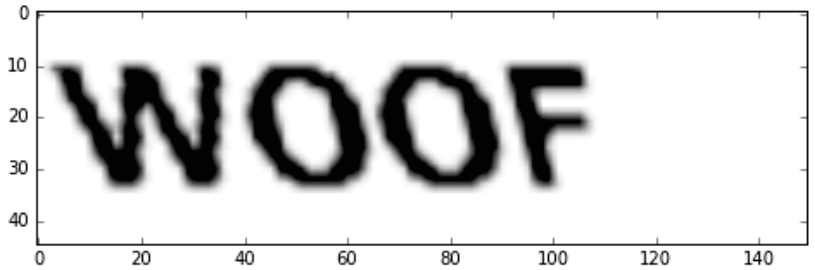



```
image = create_captcha("BARK", shear=0.8, scale=1.0)
plt.imshow(image, cmap='Greys')
```



Here is a variant scaled to `1.5` sized. While it looks similar to the BONE image above, note the *x*-axis and *y*-axis values are larger:

```
image = create_captcha("WOOF", shear=0.25, scale=1.5)
plt.imshow(image, cmap='Greys')
```



Splitting the image into individual letters

Our CAPTCHAs are words. Instead of building a classifier that can identify the thousands and thousands of possible words, we will break the problem down into a smaller problem: predicting letters.

Our experiment is in English, and all uppercase, meaning we have 26 classes to predict from for each letter. If you try these experiments in other languages, keep in mind the number of output classes will have to change.

The first step in our algorithm for beating these CAPTCHAs involves segmenting the word to discover each of the letters within it. To do this, we are going to create a function that finds contiguous sections of black pixels in the image and extract them as subimages. These are (or at least should be) our letters. The `scikit-image` function has tools for performing these operations.

Our function will take an image, and return a list of sub-images, where each sub-image is a letter from the original word in the image. The first thing we need to do is to detect where each letter is. To do this, we will use the label function in `scikit-image`, which finds connected sets of pixels that have the same value. This has analogies to our connected component discovery in `Chapter 7`, *Follow Recommendations Using Graph Mining*.

```
from skimage.measure import label, regionprops

def segment_image(image):
    # label will find subimages of connected non-black pixels
    labeled_image = label(image>0.2, connectivity=1, background=0)
    subimages = []
    # regionprops splits up the subimages
    for region in regionprops(labeled_image):
        # Extract the subimage
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x,start_y:end_y])
        if len(subimages) == 0:
            # No subimages found, so return the entire image
            return [image,]
    return subimages
```

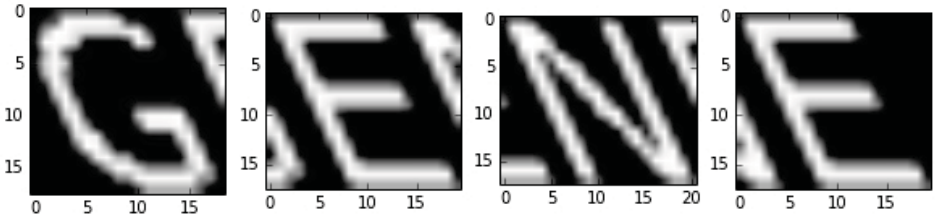
We can then get the subimages from the example CAPTCHA using this function:

```
subimages = segment_image(image)
```

We can also view each of these subimages:

```
f, axes = plt.subplots(1, len(subimages), figsize=(10, 3))
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap="gray")
```

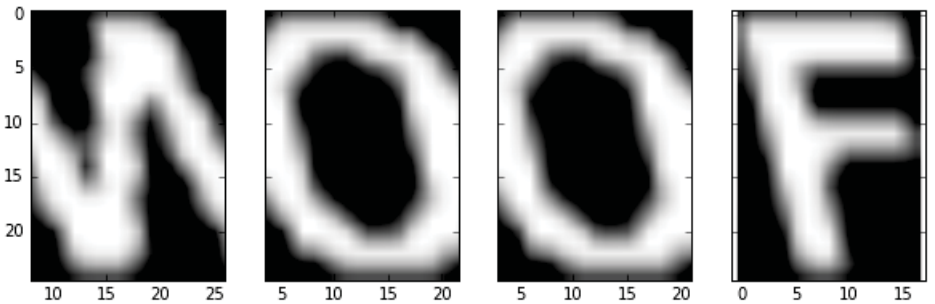
The result will look something like this:



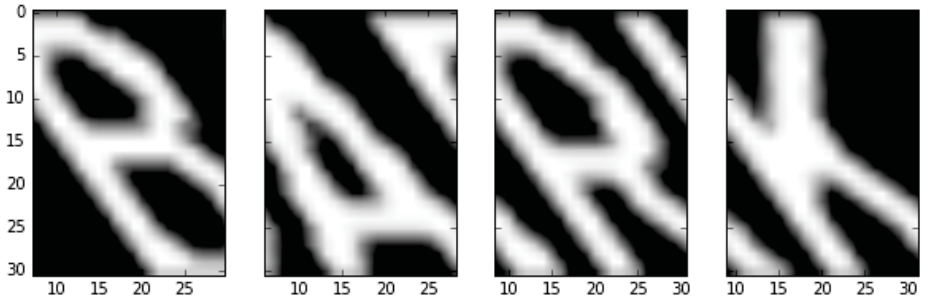
As you can see, our image segmentation does a reasonable job, but the results are still quite messy, with bits of previous letters showing. This is fine, and almost preferable. While training on data with regular noise makes our training worse, training on data with random noise can actually make it better. One reason is that the underlying data mining model learns the important aspects, namely the non-noise parts instead of specific noise inherent in the training data set. It is a fine line between too much and too little noise, and this can be hard to properly model. Testing on validation sets is a good way to ensure your training is improving.

One important note is that this code is not consistent in finding letters. Lower shear values typically result in accurately segmented images. For example, here is the code to segment the WOOF example from above:

```
image = create_captcha("WOOF", shear=0.25, scale=1.5)
subimages = segment_image(image)
f, axes = plt.subplots(1, len(subimages), figsize=(10, 3), sharey=True)
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap="gray")
```



In contrast, higher shear values are not segmented properly. For example, here is the BARK example from before:



Notice the large overlap caused by the square segmentation. One suggestion for an improvement on this chapter's code is to improve our segmentation by finding non-square segments.

Creating a training dataset

Using the functions we have already defined, we can now create a dataset of letters, each with different shear values. From this, we will train a neural network to recognize each letter from the image.

We first set up our random state and an array that holds the options for letters, shear values and scale values that we will randomly select from. There isn't much surprise here, but if you haven't used NumPy's `arange` function before, it is similar to Python's `range` function—except this one works with NumPy arrays and allows the step to be a float. The code is as follows:

```
from sklearn.utils import check_random_state
random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)
scale_values = np.arange(0.5, 1.5, 0.1)
```

We then create a function (for generating a single sample in our training dataset) that randomly selects a letter, a shear value, and a scale value selected from the available options.

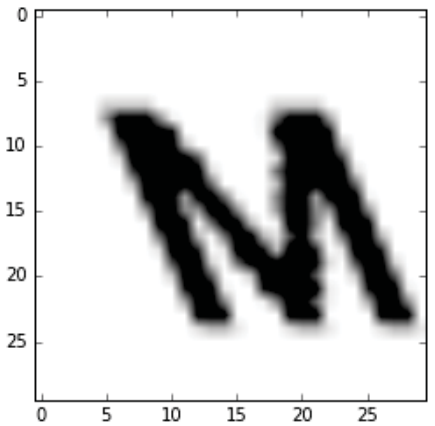
```
def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
    scale = random_state.choice(scale_values)
    # We use 30,30 as the image size to ensure we get all the text in the
image
    return create_captcha(letter, shear=shear, size=(30, 30), scale=scale),
letters.index(letter)
```

We return the image of the letter, along with the target value representing the letter in the image. Our classes will be 0 for A, 1 for B, 2 for C, and so on.

Outside the function block, we can now call this code to generate a new sample and then show it using `pyplot`:

```
image, target = generate_sample(random_state)
plt.imshow(image, cmap="Greys")
print("The target for this image is: {0}".format(target))
```

The resulting image has just a single letter, with a random shear and random scale value.



We can now generate all of our data by calling this several thousand times. We then put the data into NumPy arrays, as they are easier to work with than lists. The code is as follows:

```
dataset, targets = zip(*(generate_sample(random_state) for i in
range(1000)))
dataset = np.array([tf.resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
dataset = np.array(dataset, dtype='float')
targets = np.array(targets)
```

Our targets are integer values between 0 and 26, with each representing a letter of the alphabet. Neural networks don't usually support multiple values from a single neuron, instead preferring to have multiple outputs, each with values 0 or 1. We perform one-hot-encoding of the targets, giving us a target array that has 26 outputs per sample, using values near 1 if that letter is likely and near 0 otherwise. The code is as follows:

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
```

From this output, we know that our neural network's output layer will have 26 neurons. The goal of the neural network is to determine which of these neurons to fire, based on a given input--the pixels that compose the image.

The library we are going to use doesn't support sparse arrays, so we need to turn our sparse matrix into a dense NumPy array. The code is as follows:

```
y = y.todense()
X = dataset.reshape((dataset.shape[0], dataset.shape[1] *
dataset.shape[2]))
```

Finally, we perform a train/test split to later evaluate our data:

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.9)
```

Training and classifying

We are now going to build a neural network that will take an image as input and try to predict which (single) letter is in the image.

We will use the training set of single letters we created earlier. The dataset itself is quite simple. We have a 20-by-20-pixel image, each pixel 1 (black) or 0 (white). These represent the 400 features that we will use as inputs into the neural network. The outputs will be 26 values between 0 and 1, where higher values indicate a higher likelihood that the associated letter (the first neuron is A, the second is B, and so on) is the letter represented by the input image.

We are going to use the scikit-learn's `MLPClassifier` for our neural network in this chapter.

You will need a recent version of `scikit-learn` to use MLPClassifier. If the below import statement fails, try again after updating scikit-learn. You can do this using the following Anaconda command:

```
conda update scikit-learn
```

As for other `scikit-learn` classifiers, we import the model type and create a new one. The constructor below specifies that we create one hidden layer with 100 nodes in it. The size of the input and output layers is determined at training time:

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(100,), random_state=14)
```

To see the internal parameters of the neural network, we can use the `get_params()` function. This function exists on all `scikit-learn` models. Here is the output from the above model. Many of these parameters can improve training or the speed of training. For example, increasing the learning rate will train the model faster, at the risk of missing optimal values:

```
{'activation': 'relu',
 'alpha': 0.0001,
 'batch_size': 'auto',
 'beta_1': 0.9,
 'beta_2': 0.999,
 'early_stopping': False,
 'epsilon': 1e-08,
 'hidden_layer_sizes': (100,),
 'learning_rate': 'constant',
 'learning_rate_init': 0.001,
 'max_iter': 200,
 'momentum': 0.9,
 'nesterovs_momentum': True,
 'power_t': 0.5,
 'random_state': 14,
 'shuffle': True,
 'solver': 'adam',
```

```
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': False,
 'warm_start': False}
```

Next, we fit our model using the standard scikit-learn interface:

```
clf.fit(X_train, y_train)
```

Our model has now learned weights between each of the layers. We can view those weights by examining `clf.coefs_`, which is a list of NumPy arrays that join each of the layers. For example, the weights between the input layer with 400 neurons (from each of our pixels) to the hidden layer with 100 neurons (a parameter we set), can be obtained using `clf.coefs_[0]`. In addition, the weights between the hidden layer and the output layer (with 26 neurons) can be obtained using `clf.coefs_[1]`. These weights, together with the parameters above, wholly define our trained network.

We can now use that trained network to predict our test dataset:

```
y_pred = clf.predict(X_test)
```

Finally, we evaluate the results:

```
from sklearn.metrics import f1_score
f1_score(y_pred=y_pred, y_true=y_test, average='macro')
```

The result is 0.96, which is pretty impressive. This version of the F1 score is based on the macro-average, which computes the individual F1 score for each class, and then averages them without considering the size of each class.

To examine these individual class results, we can view the classification report:

```
from sklearn.metrics import classification_report
print(classification_report(y_pred=y_pred, y_true=y_test))
```

The results from my experiment are shown here:

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         5
           1       1.00      1.00      1.00         3
           2       1.00      1.00      1.00         3
           3       1.00      1.00      1.00         8
           4       1.00      1.00      1.00         2
           5       1.00      1.00      1.00         4
           6       1.00      1.00      1.00         2
           7       1.00      1.00      1.00         2
```

```
              8       1.00      1.00      1.00         7
              9       1.00      1.00      1.00         1
             10       1.00      1.00      1.00         3
             11       1.00      1.00      1.00         4
             12       1.00      0.75      0.86         4
             13       1.00      1.00      1.00         5
             14       1.00      1.00      1.00         4
             15       1.00      1.00      1.00         3
             16       1.00      1.00      1.00         3
             17       1.00      1.00      1.00         7
             18       1.00      1.00      1.00         5
             19       1.00      1.00      1.00         5
             20       1.00      1.00      1.00         3
             21       1.00      1.00      1.00         5
             22       1.00      1.00      1.00         2
             23       1.00      1.00      1.00         4
             24       1.00      1.00      1.00         2
             25       1.00      1.00      1.00         4

avg / total       1.00      0.99      0.99       100
```

The final `f1-score` for this report is shown on the bottom right, the second last number - 0.99. This is the micro-average, where the `f1-score` is computed for each sample and then the mean is computed. This form makes more sense for relatively similar class sizes, while the macro-average makes more sense for imbalanced classes.

Pretty simple from an API perspective, as `scikit-learn` hides all of the complexity. However what actually happened in the backend? How do we train a neural network?

Back-propagation

Training a neural network is specifically focused on the following things.

- The first is the size and shape of the network - how many layers, what sized layers and what error functions they use. While types of neural networks exists that can alter their size and shape, the most common type, a feed-forward neural network, rarely has this capability. Instead, its size is fixed at initialization time, which in this chapter is 400 neurons in the first layer, 100 in the hidden layer and 26 in the final layer. Training for the shape is usually the job of a meta-algorithm that trains a set of neural networks and determines which is the most effective, outside of training the networks themselves.

- The second part of training a neural network is to alter the weights between neurons. In a standard neural network, nodes from one layer are attached to nodes of the next layer by edges with a specific weight. These can be initialized randomly (although several smarter methods do exist such as autoencoders), but need to be adjusted to allow the network to *learn* the relationship between training samples and training classes.

This adjusting of weights was one of the key issues holding back very-early neural networks, before an algorithm called **back propagation** was developed to solve the issue.

The **back propagation** (**backprop**) algorithm is a way of assigning blame to each neuron for incorrect predictions. First, we consider the usage of a neural network, where we feed a sample into the input layer and see which of the output layer's neurons fire, as *forward propagation*. Back propagation goes backwards from the output layer to the input layer, assigning blame to each weight in the network, in proportion to the effect that weight has on any errors that the network makes.

The amount of change is based on two aspects:

- Neuron activation
- The gradient of the activation function

The first is the degree to which the neuron was *activated*. Neurons that fire with high (absolute) values are considered to have a great impact on the result, while those that fired with small (absolute) values have a low impact on the result. Due to this, weights around neurons that fire with high values are changed more than those around small values.

The second aspect to the amount that weights change is proportional to the *gradient of the activation function*. Many neural networks you use will have the same activation function for all neurons, but there are lots of situations where it makes sense to have different activation functions for different layers of neurons (or more rarely, different activation functions in the same layer). The gradient of the activation function, combined with the activation of the neuron, and the error assigned to that neuron, together form the amount that the weights are changed.

I've skipped over the maths involved in back propagation, as the focus of this book is on practical usage. As you increase your usage of neural networks, it pays to know more about what goes on inside the algorithm. I recommend looking into the details of the back-prop algorithm, which can be understood with some basic knowledge of gradients and derivatives.

Predicting words

Now that we have a classifier for predicting individual letters, we now move onto the next step in our plan - predicting words. To do this, we want to predict each letter from each of these segments, and put those predictions together to form the predicted word from a given CAPTCHA.

Our function will accept a CAPTCHA and the trained neural network, and it will return the predicted word:

```
def predict_captcha(captcha_image, neural_network):
    subimages = segment_image(captcha_image)
    # Perform the same transformations we did for our training data
    dataset = np.array([np.resize(subimage, (20, 20)) for subimage in
subimages])
    X_test = dataset.reshape((dataset.shape[0], dataset.shape[1] *
dataset.shape[2]))
    # Use predict_proba and argmax to get the most likely prediction
    y_pred = neural_network.predict_proba(X_test)
    predictions = np.argmax(y_pred, axis=1)

    # Convert predictions to letters
    predicted_word = str.join("", [letters[prediction] for prediction in
predictions])
    return predicted_word
```

We can now test on a word using the following code. Try different words and see what sorts of errors you get, but keep in mind that our neural network only knows about capital letters:

```
word = "GENE"
captcha = create_captcha(word, shear=0.2)
print(predict_captcha(captcha, clf))
plt.imshow(captcha, cmap="Greys")
```

We can codify this into a function, allowing us to perform predictions more easily:

```
def test_prediction(word, net, shear=0.2, scale=1):
    captcha = create_captcha(word, shear=shear, scale=scale,
size=(len(word) * 25, 30))
    prediction = predict_captcha(captcha, net)
    return word == prediction, word, prediction
```

The returned results specify whether the prediction is correct, the original word, and the predicted word. This code correctly predicts the word GENE, but makes mistakes with other words. How accurate is it? To test, we will create a dataset with a whole bunch of four-letter English words from NLTK. The code is as follows:

```
from nltk.corpus import words
```

Install NLTK using Anaconda: **conda install nltk**
After installation, and before using it in code, you will need to download the corpus using:
python -c "import nltk; nltk.download('words')"

The words instance here is actually a corpus object, so we need to call `words()` on it to extract the individual words from this corpus. We also filter to get only four-letter words from this list:

```
valid_words = set([word.upper() for word in words.words() if len(word) ==
4])
```

We can then iterate over all of the words to see how many we get correct by simply counting the correct and incorrect predictions:

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    shear = random_state.choice(shear_values)
    scale = random_state.choice(scale_values)
    correct, word, prediction = test_prediction(word, clf, shear=shear,
scale=scale)
    if correct:
        num_correct += 1
    else:
        num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

The results I get are 3,342 correct and 2,170 incorrect for an accuracy of just over 62 percent. From our original 99 percent per-letter accuracy, this is a big decline. What happened?

The reasons for this decline are listed here:

- The first factor to impact is our accuracy. All other things being equal, if we have four letters, and 99 percent accuracy per-letter, then we can expect about a 96 percent success rate (all other things being equal) getting four letters in a row (0.99^4≈0.96). A single error in a single letter's prediction results in the wrong word being predicted.
- The second impact is the shear value. Our dataset chose randomly between shear values of 0 to 0.5. The previous test used a shear of 0.2. For a value of 0, I get 75 percent accuracy; for a shear of 0.5, the result is much worse at 2.5 percent. The higher the shear, the lower the performance.
- The third impact is that often words are incorrectly segmented. Another issue is that some vowels are commonly mistaken, causing more errors than can be expected by the above error rates.

Let's examine the second of these issues, and map the relationship between shear and performance. First, we turn our evaluation code into a function that is dependent on a given shear value:

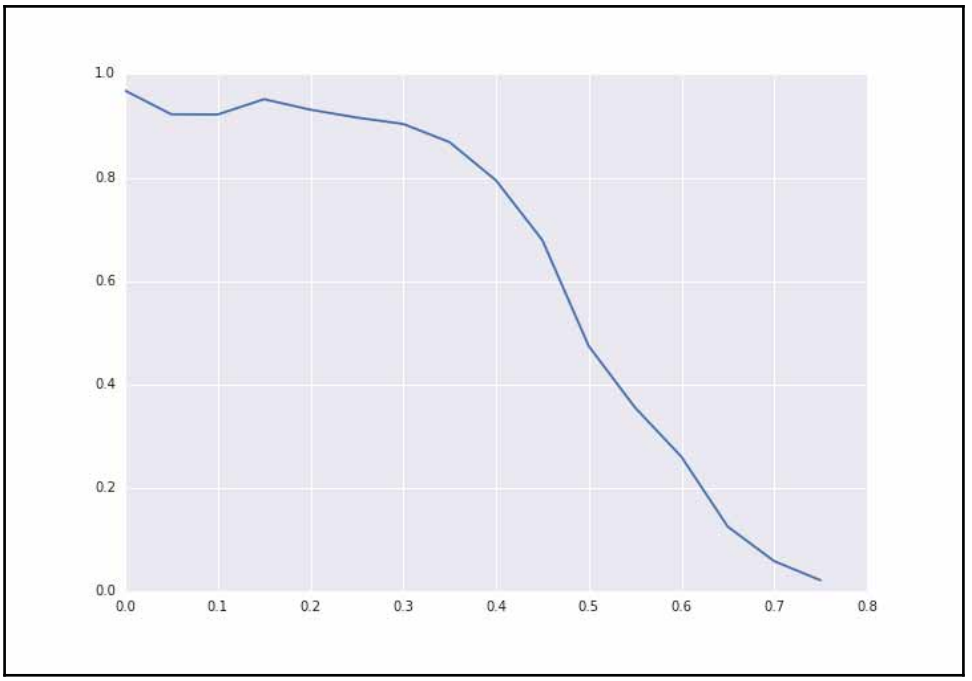
```
def evaluation_versus_shear(shear_value):
    num_correct = 0
    num_incorrect = 0
    for word in valid_words:
        scale = random_state.choice(scale_values)
        correct, word, prediction = test_prediction(word, clf,
shear=shear_value, scale=scale)
        if correct:
            num_correct += 1
        else:
            num_incorrect += 1
    return num_correct/(num_correct+num_incorrect)
```

Then, we take a list of shear values and then use this function to evaluate the accuracy for each value. Note that this code will take a while to run, approximately 30 minutes depending on your hardware.

```
scores = [evaluation_versus_shear(shear) for shear in shear_values]
```

Finally, plot the result using matplotlib:

```
plt.plot(shear_values, scores)
```



You can see that there is a severe drop in performance as the shear value increases past 0.4. Normalizing the input would help, with tasks such as image rotation and unshearing the input.

Another surprising option to address issues with shear is to increase the amount of training data with high shear values, which can lead to the model learning a more generalized output.

We look into improving the accuracy using post-processing in the next section.

Improving accuracy using a dictionary

Rather than just returning the given prediction, we can check whether the word actually exists in our dictionary. If it does, then that is our prediction. If it isn't in the dictionary, we can try and find a word that is similar to it and predict that instead. Note that this strategy relies on our assumption that all CAPTCHA words will be valid English words, and therefore this strategy wouldn't work for a random sequence of characters. This is one reason why some CAPTCHAs don't use words.

There is one issue here—how do we determine the closest word? There are many ways to do this. For instance, we can compare the lengths of words. Two words that have a similar length could be considered more similar. However, we commonly consider words to be similar if they have the same letters in the same positions. This is where the **edit distance** comes in.

Ranking mechanisms for word similarity

The **Levenshtein edit distance** is a commonly used method for comparing two short strings to see how similar they are. It isn't very scalable, so it isn't commonly used for very long strings. The edit distance computes the number of steps it takes to go from one word to another. The steps can be one of the following three actions:

- Insert a new letter into the word at any position
- Delete any letter from the word
- Substitute a letter for another one

The minimum number of actions needed to transform the first word into the second is given as the distance. Higher values indicate that the words are less similar.

This distance is available in NLTK as `nltk.metrics.edit_distance`. We can call it using only two strings and it returns the edit distance:

```
from nltk.metrics import edit_distance
steps = edit_distance("STEP", "STOP")
print("The number of steps needed is: {0}".format(steps))
```

When used with different words, the edit distance is quite a good approximation to what many people would intuitively feel are similar words. The edit distance is great for testing spelling mistakes, dictation errors, and name matching (where you can mix up your Marc and Mark spelling quite easily).

However, it isn't very good for our case. We don't really expect letters to be moved around, just individual letter comparisons to be wrong. For this reason, we will create a different distance metric, which is simply the number of letters in the same positions that are incorrect. The code is as follows:

```
def compute_distance(prediction, word):
    len_word = min(len(prediction), len(word))
    return len_word - sum([prediction[i] == word[i] for i in
range(len_word)])
```

We subtract the value from the length of the prediction word (which is four) to make it a distance metric where lower values indicate more similarity between the words.

Putting it all together

We can now test our improved prediction function using similar code to before. First, we define a prediction function, which also takes our list of valid words:

```
from operator import itemgetter

def improved_prediction(word, net, dictionary, shear=0.2, scale=1.0):
    captcha = create_captcha(word, shear=shear, scale=scale)
    prediction = predict_captcha(captcha, net)

    if prediction not in dictionary:
        distances = sorted([(word, compute_distance(prediction, word)) for
word in dictionary],
                           key=itemgetter(1))
        best_word = distances[0]
        prediction = best_word[0]
    return word == prediction, word, prediction
```

We compute the distance between our predicted word and each other word in the dictionary, and sort it by distance (lowest first). The changes in our testing code are in the following code:

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    shear = random_state.choice(shear_values)
    scale = random_state.choice(scale_values)
    correct, word, prediction = improved_prediction(word, clf, valid_words,
shear=shear, scale=scale)
    if correct:
        num_correct += 1
```

```
        else:
            num_incorrect += 1
    print("Number correct is {0}".format(num_correct))
    print("Number incorrect is {0}".format(num_incorrect))
```

The preceding code will take a while to run (computing all of the distances will take some time) but the net result is 3,037 samples correct and 2,476 samples incorrect. This is an accuracy of 71.5 percent for a boost of nearly 10 percentage points!

Looking for a challenge? Update the `predict_captcha` function to return the probabilities assigned to each letter. By default, the letter with the highest probability is chosen for each letter in a word. If that doesn't work, choose the next most probable word, by multiplying the per-letter probabilities together.

Summary

In this chapter, we worked with images in order to use simple pixel values to predict the letter being portrayed in a CAPTCHA. Our CAPTCHAs were a bit simplified; we only used complete four-letter English words. In practice, the problem is much harder--as it should be! With some improvements, it would be possible to solve much harder CAPTCHAs with neural networks and a methodology similar to what we discussed. The `scikit-image` library contains lots of useful functions for extracting shapes from images, functions for improving contrast, and other image tools that will help.

We took our larger problem of predicting words, and created a smaller and simple problem of predicting letters. From here, we were able to create a feed-forward neural network to accurately predict which letter was in the image. At this stage, our results were very good with 97 percent accuracy.

Neural networks are simply connected sets of neurons, which are basic computation devices consisting of a single function. However, when you connect these together, they can solve incredibly complex problems. Neural networks are the basis for deep learning, which is one of the most effective areas of data mining at the moment.

Despite our great per-letter accuracy, the performance when predicting a word drops to just over 60 percent when trying to predict a whole word. We improved our accuracy using a dictionary, searching for the best matching word. To do this, we considered the commonly used edit distance; however, we simplified it because we were only concerned with individual mistakes on letters, not insertions or deletions. This improvement netted some benefit, but there are still many improvements you could try to further boost the accuracy.

To take the concepts in this chapter further, investigate changing the neural network structure, by adding more hidden layers, or changing the shape of those layers. Investigate the impact this has on the result. Further, try creating a more difficult CAPTCHA--does this drop the accuracy? Can you build a more complicated network to learn it?

Data mining problems such as the CAPTCHA example show that an initial problem statement, such as *guess this word*, can be broken into individual subtasks that can be performed using data mining. Further, those subtasks can be combined in a few different ways, such as with the use of external information. In this chapter, we combined our letter prediction with a dictionary of valid words to provide a final response, giving better accuracy than letter prediction alone.

In the next chapter, we will continue with string comparisons. We will attempt to determine which author (out of a set of authors) wrote a particular document--using only the content and no other information!

9
Authorship Attribution

Authorship analysis is a text mining task that aims to identify certain aspects about an author, based only on the content of their writings. This could include characteristics such as age, gender, or background. In the specific **authorship attribution** task, we aim to identify which of a set of authors wrote a particular document. This is a classic classification task. In many ways, authorship analysis tasks are performed using standard data mining methodologies, such as cross-fold validation, feature extraction, and classification algorithms.

In this chapter, we will use the problem of authorship attribution to piece together the parts of the data mining methodology we developed in the previous chapters. We identify the problem and discuss the background and knowledge of the problem. This lets us choose features to extract, which we will build a pipeline for achieving. We will test two different types of features: function words and character n -grams. Finally, we will perform an in-depth analysis of the results. We will work first with a dataset of books, and then a messy, real-world corpus of e-mails.

The topics we will cover in this chapter are as follows:

- Feature engineering and how feature choice differs based on application
- Revisiting the bag-of-words model with a specific goal in mind
- Feature types and the character n-grams model
- Support Vector Machines
- Cleaning up a messy dataset for data mining

Attributing documents to authors

Authorship analysis has a background in **stylometry**, which is the study of an author's style of writing. The concept is based on the idea that everyone learns language slightly differently, and that measuring these nuances in people's writing will enable us to tell them apart using only the content of their writing.

Authorship analysis has historically (pre-1990) been performed using repeatable manual analysis and statistics, which is a good indication that it could be automated with data mining. Modern authorship analysis studies are almost entirely data mining-based, although quite a significant amount of work is still done with more manually driven analysis using linguistic styles and stylometrics. Many of the advances in feature engineering today are driven by advances in stylometrics. In other words, manual analysis discovers new features, which are then codified and used as part of the data mining process.

A key underlying feature of stylometry is that of **writer invariants**, which are features that a particular author has in all of their documents, but are not shared with other authors. In practice these writer invariants do not seem to exist, as authorship styles change over time, but the use of data mining can get us close to classifiers working off this principle.

As a field, authorship analysis has many sub-problems, and the main ones are as follows:

- **Authorship profiling:** This determines the age, gender, or other traits of the author based on the writing. For example, we can detect the first language of a person speaking English by looking for specific ways in which they speak the language.
- **Authorship verification:** This checks whether the author of this document also wrote the other document. This problem is what you would normally think about in a legal court setting. For instance, the suspect's writing style (content-wise) would be analyzed to see if it matched the ransom note.
- **Authorship clustering:** This is an extension of authorship verification, where we use cluster analysis to group documents from a big set into clusters, and each cluster is written by the same author.

However, the most common form of authorship analysis study is that of **authorship attribution**, a classification task where we attempt to predict which of a set of authors wrote a given document.

Applications and use cases

Authorship analysis has a number of **use cases**. Many use cases are concerned with problems such as verifying authorship, proving shared authorship/provenance, or linking social media profiles with real-world users.

In a historical sense, we can use authorship analysis to verify whether certain documents were indeed written by their supposed authors. Controversial authorship claims include some of Shakespeare's plays, the Federalist papers from the USA's foundation period, and other historical texts.

Authorship studies alone cannot prove authorship but can provide evidence for or against a given theory, such as whether a particular person wrote a given document.

For example, we can analyze Shakespeare's plays to determine his writing style, before testing whether a given sonnet actually does originate from him (some recent research indicates multiple authorship of some of his work).

A more modern use case is that of linking social network accounts. For example, a malicious online user could set up accounts on multiple online social networks. Being able to link them allows authorities to track down the user of a given account—for example if a person is harassing other online users.

Another example used in the past is to be a backbone to provide expert testimony in court to determine whether a given person wrote a document. For instance, the suspect could be accused of writing an e-mail harassing another person. The use of authorship analysis could determine whether it is likely that person did, in fact, write the document. Another court-based use is to settle claims of stolen authorship. For example, two authors may claim to have written a book, and authorship analysis could provide evidence on which is the more likely author.

Authorship analysis is not foolproof, though. A recent study found that attributing documents to authors can be made considerably harder by simply asking people, who are otherwise untrained, to hide their writing style. This study also looked at a framing exercise where people were asked to write in the style of another person. This framing of another person proved quite reliable, with the faked document commonly attributed to the person being framed.

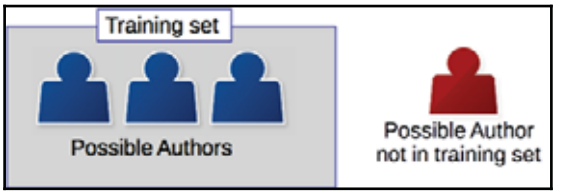
Despite these issues, authorship analysis is proving useful in a growing number of areas and is an interesting data mining problem to investigate.

Authorship attribution can be used in expert testimony, but by itself is hard to classify as hard evidence. Always check with a lawyer before using it for formal matters, such as authorship disputes.

Authorship attribution

Authorship attribution (as distinct from authorship *analysis*) is a classification task by which we have a set of candidate authors, a set of documents from each of those authors namely the **training set**, and a set of documents of unknown authorship otherwise known as the test set. If the documents of unknown authorship definitely belong to one of the candidates, we call this a closed problem, as per the following diagram:

If we cannot be sure of that the actual author is part of the training set, we call this an open problem. This distinction isn't just specific to authorship attribution - any data mining application where the actual class may not be in the training set is considered an open problem, with the task being to find the candidate author or to select none of them. This is shown in the following diagram:



In authorship attribution, we typically have two restrictions on the tasks. They have been listed as follows:

- First, we only use content information from the documents - not metadata regarding the time of writing, delivery, handwriting style, and so on. There are ways to combine models from these different types of information, but that isn't generally considered authorship attribution and is more a **data fusion** application.
- The second restriction is that we don't look at the topic of the documents; instead, we look for more salient features such as word usage, punctuation, and other text-based features. The reasoning here is that a person can write on many different topics, so worrying about the topic of their writing isn't going to model their actual authorship style. Looking at topic words can also lead to **overfitting** on the training data—our model may train on documents from the same author and also on the same topic. For instance, if you were to model my authorship style by looking at this book, you might conclude the words *data mining* are indicative of *my* writing style when, in fact, I write on other topics as well.

From here, the pipeline for performing authorship attribution looks a lot like the one we developed in `Chapter 6`, *Social Media Insight Using Naive Bayes*.

1. First, we extract features from our text.
2. Then, we perform some feature selection on those features.
3. Finally, we train a classification algorithm to fit a model, which we can then use to predict the class (in this case, the author) of a document.

There are some differences between classifying content and classifying authorship, mostly having to do with which features are used, that we will cover in this chapter. It is critical to choose features based on the application.

Before we delve into these issue, we will define the scope of the problem and collect some data.

Getting the data

The data we will use for the first part of this chapter is a set of books from **Project Gutenberg** at `www.gutenberg.org`, which is a repository of public domain literature works. The books I used for these experiments come from a variety of authors:

- Booth Tarkington (22 titles)
- Charles Dickens (44 titles)
- Edith Nesbit (10 titles)
- Arthur Conan Doyle (51 titles)
- Mark Twain (29 titles)
- Sir Richard Francis Burton (11 titles)
- Emile Gaboriau (10 titles)

Overall, there are 177 documents from 7 authors, giving a significant amount of text to work with. A full list of the titles, along with download links and a script to automatically fetch them, is given in the code bundle called **getdata.py**. If running the code results in significantly fewer books than above, the mirror may be down. See this website for more mirror URLs to try in the script: `https://www.gutenberg.org/MIRRORS.ALL`

To download these books, we use the requests library to download the files into our data directory.

First, in a new Jupyter Notebook, set up the data directory and ensure the following code links to it:

```
import os
import sys
data_folder = os.path.join(os.path.expanduser("~"), "Data", "books")
```

Next, download the data bundle from the code bundle supplied by Packt. Decompress the file into this directory. The books folder should then directly contain one folder for each author.

After taking a look at these files, you will see that many of them are quite messy—at least from a data analysis point of view. There is a large project Gutenberg disclaimer at the start of the files. This needs to be removed before we do our analysis.

For example, most books begin with information such as the following:

The Project Gutenberg eBook of Mugby Junction, by Charles Dickens, et al, Illustrated by Jules A. Goodman This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at www.gutenberg.org
Title: Mugby Junction
Author: Charles Dickens
Release Date: January 28, 2009 [eBook #27924]Language: English
Character set encoding: UTF-8
****START OF THE PROJECT GUTENBERG EBOOK MUGBY JUNCTION****

After this point, the actual text of the book starts. The use of a line starting *****START OF THE PROJECT GUTENBERG** is fairly consistent, and we will use that as a cue on when the text starts - anything before this line will be ignored.

We could alter the individual files on disk to remove this stuff. However, what happens if we were to lose our data? We would lose our changes and potentially be unable to replicate the study. For that reason, we will perform the preprocessing as we load the files—this allows us to be sure our results will be replicable (as long as the data source stays the same). The following code removes the main source of noise from the books, which is the prelude that Project Gutenberg adds to the files:

```
def clean_book(document):
    lines = document.split("n")
    start= 0
    end = len(lines)
    for i in range(len(lines)):
        line = lines[i]
        if line.startswith("*** START OF THIS PROJECT GUTENBERG"):
            start = i + 1
        elif line.startswith("*** END OF THIS PROJECT GUTENBERG"):
            end = i - 1
    return "n".join(lines[start:end])
```

You may want to add to this function to remove other sources of noise, such as inconsistent formatting, footer information, and so on. Investigate the files to examine what issues they have.

We can now get our documents and classes using the following function, which loops through these folders, loads the text documents and records a number assigned to the author as the target class.

```
import numpy as np

def load_books_data(folder=data_folder):
    documents = []
    authors = []
    subfolders = [subfolder for subfolder in os.listdir(folder)
                  if os.path.isdir(os.path.join(folder, subfolder))]
    for author_number, subfolder in enumerate(subfolders):
        full_subfolder_path = os.path.join(folder, subfolder)
        for document_name in os.listdir(full_subfolder_path):
            with open(os.path.join(full_subfolder_path, document_name),
errors='ignore') as inf:
                documents.append(clean_book(inf.read()))
                authors.append(author_number)
    return documents, np.array(authors, dtype='int')
```

We then call this function to actually load the books:

```
documents, classes = load_books_data(data_folder)
```

This dataset fits into memory quite easily, so we can load all of the text at once. In cases where the whole dataset doesn't fit, a better solution is to extract the features from each document one-at-a-time (or in batches) and save the resulting values to a file or in-memory matrix

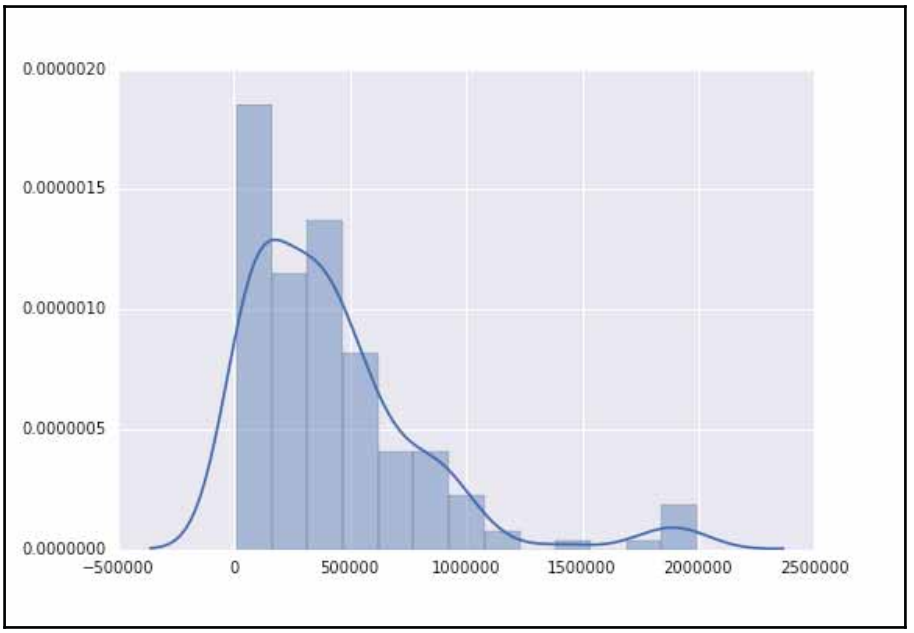
To get a gauge on the properties of the data, one of the first things I usually do is create a simple histogram of the document lengths. If the lengths are relatively consistent, this is often easier to learn from than wildly different document lengths. In this case, there is quite a large variance in document lengths. To view this, first we extract the lengths into a list:

```
document_lengths = [len(document) for document in documents]
```

Next, we plot those. Matplotlib has a `hist` function that will do this, as does Seaborn, which produces nicer looking graphs by default.

```
import seaborn as sns
sns.distplot(document_lengths)
```

The resulting graph shows the variation in document lengths:



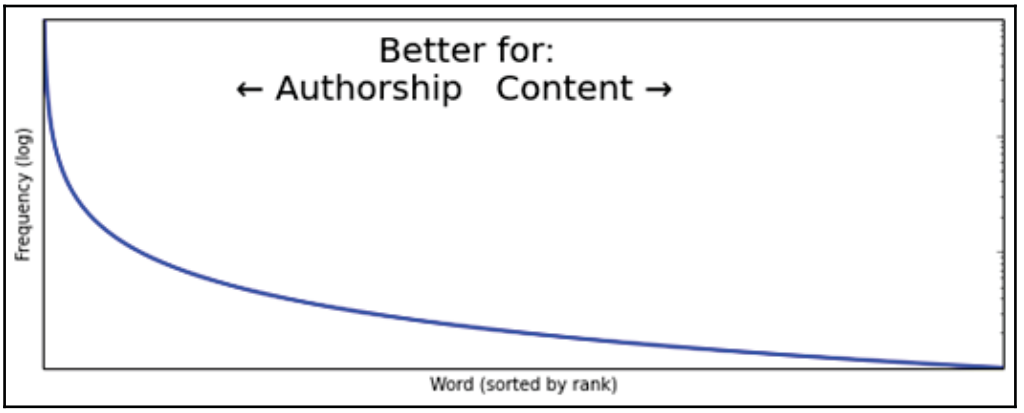
Using function words

One of the earliest types of features, and one that still works quite well for authorship analysis, is to use function words in a bag-of-words model. Function words are words that have little meaning on their own, but are required for creating (English!) sentences. For example, the words *this* and *which* are words that are really only defined by what they do within a sentence, rather than their meaning in themselves. Contrast this with a content word such as *tiger*, which has an explicit meaning and invokes imagery of a large cat when used in a sentence.

The set of words that are considered function words is not always obvious. A good rule of thumb is to choose the most frequent words in usage (over all possible documents, not just ones from the same author).

Typically, the more frequently a word is used, the better it is for authorship analysis. In contrast, the less frequently a word is used, the better it is for content-based text mining, such as in the next chapter, where we look at the topic of different documents.

The graph here gives a better idea between word and frequency relationship:



The use of function words is less defined by the content of the document and more by the decisions made by the author. This makes them good candidates for separating the authorship traits between different users. For instance, while many Americans are particular about the different in usage between *that* and *which* in a sentence, people from other countries, such as Australia, are less concerned with the distinction. This means that some Australians will lean towards almost exclusively using one word or the other, while others may use *which* much more.

This difference, combined with thousands of other nuanced differences, makes a model of authorship.

Counting function words

We can count function words using the **CountVectorizer** class we used in `Chapter 6`, *Social Media Insight Using Naive Bayes*. This class can be passed a vocabulary, which is the set of words it will look for. If a vocabulary is not passed (we didn't pass one in the code of `Chapter 6`, *Social Media Insight Using Naive Bayes*), then it will learn this vocabulary from the training dataset. All the words are in the training set of documents (depending on the other parameters of course).

First, we set up our vocabulary of function words, which is just a list containing each of them. Exactly which words are function words and which are not is up for debate. I've found the following list, from published research, to be quite good, obtained from my own research combining word lists from other researchers. Remember that the code bundle is available from Packt publishing (or the official github channel), and therefore you don't need to type this out:

```
function_words = ["a", "able", "aboard", "about", "above", "absent",
"according" , "accordingly", "across", "after", "against","ahead",
"albeit", "all", "along", "alongside", "although", "am", "amid", "amidst",
"among", "amongst", "amount", "an", "and", "another", "anti", "any",
"anybody", "anyone", "anything", "are", "around", "as", "aside",
"astraddle", "astride", "at", "away", "bar", "barring", "be", "because",
"been", "before", "behind", "being", "below", "beneath", "beside",
"besides", "better", "between", "beyond", "bit", "both", "but", "by",
"can", "certain", "circa", "close", "concerning", "consequently",
"considering", "could", "couple", "dare", "deal", "despite", "down", "due",
"during", "each", "eight", "eighth", "either", "enough", "every",
"everybody", "everyone", "everything", "except", "excepting", "excluding",
"failing", "few", "fewer", "fifth", "first", "five", "following", "for",
"four", "fourth", "from", "front", "given", "good", "great", "had", "half",
"have", "he", "heaps", "hence", "her", "hers", "herself", "him", "himself",
"his", "however", "i", "if", "in", "including", "inside", "instead",
"into", "is", "it", "its", "itself", "keeping", "lack", "less", "like",
"little", "loads", "lots", "majority", "many", "masses", "may", "me",
"might", "mine", "minority",
"minus", "more", "most", "much", "must", "my", "myself", "near", "need",
"neither", "nevertheless", "next", "nine", "ninth", "no", "nobody", "none",
"nor", "nothing", "notwithstanding", "number", "numbers", "of", "off",
"on", "once", "one", "onto", "opposite", "or", "other", "ought", "our",
"ours", "ourselves", "out", "outside", "over", "part", "past", "pending",
"per", "pertaining", "place", "plenty", "plethora", "plus", "quantities",
"quantity", "quarter", "regarding", "remainder", "respecting", "rest",
"round", "save", "saving", "second", "seven", "seventh", "several","shall",
"she", "should", "similar", "since", "six", "sixth", "so", "some",
"somebody", "someone", "something", "spite","such", "ten", "tenth", "than",
"thanks", "that", "the", "their", "theirs", "them", "themselves", "then",
"thence", "therefore", "these", "they", "third", "this", "those","though",
"three", "through", "throughout", "thru", "thus", "till", "time", "to",
"tons", "top", "toward", "towards", "two", "under", "underneath", "unless",
"unlike", "until", "unto", "up", "upon", "us", "used", "various",
"versus","via", "view", "wanting", "was", "we", "were", "what", "whatever",
"when", "whenever", "where", "whereas", "wherever", "whether", "which",
"whichever", "while","whilst", "who", "whoever", "whole", "whom",
"whomever", "whose", "will", "with", "within", "without", "would", "yet",
"you", "your", "yours", "yourself", "yourselves"]
```

Now, we can set up an extractor to get the counts of these function words. Note the passing of the function words list as the `vocabulary` into the `CountVectorizer` initialiser.

```
from sklearn.feature_extraction.text
import CountVectorizer
extractor = CountVectorizer(vocabulary=function_words)
```

For this set of function words, the frequency within these documents is very high - as you would expect. We can use the extractor instance to obtain these counts, by fitting it on the data, and then calling `transform` (or, the shortcut using `fit_transform`).

```
extractor.fit(documents)
counts = extractor.transform(documents)
```

Before plotting, we normalized these counts by dividing by the relevant document lengths. The following code does this, resulting in the percentage of words accounted for by each function word:

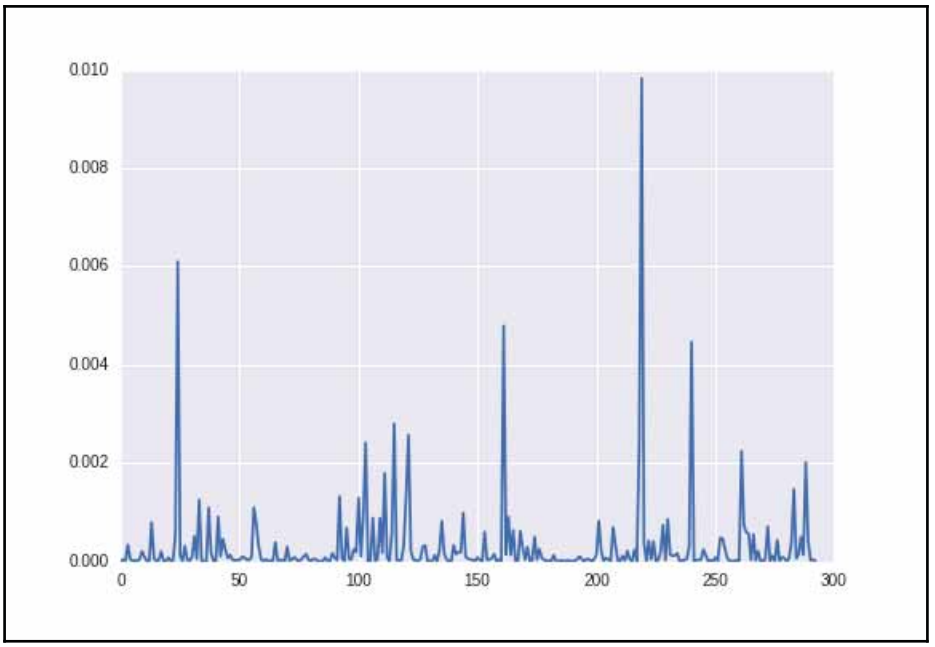
```
normalized_counts = counts.T / np.array(document_lengths)
```

We then average these percentages across all documents:

```
averaged_counts = normalized_counts.mean(axis=1)
```

Finally we plot them using Matplotlib (Seaborn lacks easy interfaces to basic plots like this).

```
from matplotlib import pyplot as plt
plt.plot(averaged_counts)
```



Classifying with function words

The only new thing here is the use of **Support Vector Machines** (**SVM**), which we will cover in the next section (for now, just consider it a standard classification algorithm).

Next, we import our classes. We import the **SVC** class, an SVM for classification, as well as the other standard workflow tools we have seen before:

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline from sklearn import grid_search
```

SVMs take a number of parameters. As I said, we will use one blindly here, before going into detail in the next section. We then use a dictionary to set which parameters we are going to search. For the `kernel` parameter, we will try `linear` and `rbf`. For **C**, we will try values of 1 and 10 (descriptions of these parameters are covered in the next section). We then create a grid search to search these parameters for the best choices:

```
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svr = SVC()
grid = grid_search.GridSearchCV(svr, parameters)
```

Gaussian kernels (such as RBF) only work for reasonably sized data sets, such as when the number of features is fewer than about 10,000.

Next, we set up a pipeline that takes the feature extraction step using the `CountVectorizer` (only using function words), along with our grid search using SVM. The code is as follows:

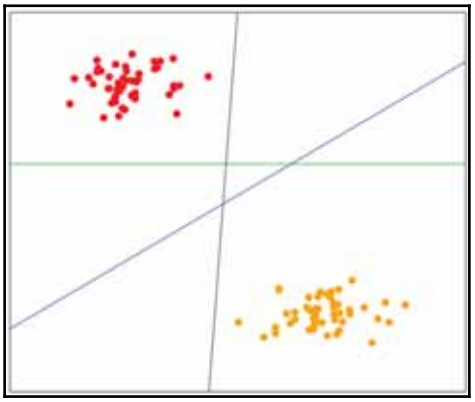
```
pipeline1 = Pipeline([('feature_extraction', extractor), ('clf', grid) ])
```

Next, apply `cross_val_score` to get our cross-validated score for this pipeline. The result is 0.811, which means we approximately get 80 percent of the predictions correct.

Support Vector Machines

SVMs are classification algorithms based on a simple and intuitive idea, backed by some complex and innovative mathematics. SVMs perform classification between two classes (although we can extend it to more classes using various meta-algorithms), by simply drawing a separating line between the two (or a hyperplane in higher-dimensions). The intuitive idea is to choose the best line of separation, rather than just any specific line.

Suppose that our two classes can be separated by a line such that any points above the line belong to one class and any below the line belong to the other class. SVMs find this line and use it for prediction, much the same way as linear regression works. SVMs, however, find the best line for separating the dataset. In the following figure, we have three lines that separate the dataset: blue, black, and green. Which would you say is the best option?



Intuitively, a person would normally choose the blue line as the best option, as this separates the data in the cleanest way. More formally, it has the maximum distance from the line to any point in each class. Finding this line of maximum separation is an optimization problem, based on finding the lines of margin with the maximum distance between them. Solving this optimisation problem is the main task of the training phase of an SVM.

The equations to solve SVMs is outside the scope of this book, but I recommend interested readers to go through the derivations at: `http://en.wikibooks.org/wiki/Support_Vector_Machines`for the details.
Alternatively, you can visit:
`http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html`

Classifying with SVMs

After training the model, we have a line of maximum margin. The classification of new samples is then simply asking the question: does it fall above the line, or below it? If it falls above the line, it is predicted as one class. If it is below the line, it is predicted as the other class.

For multiple classes, we create multiple SVMs—each a binary classifier. We then connect them using any one of a variety of strategies. A basic strategy is to create a one-versus-all classifier for each class, where we train using two classes—the given class and all other samples. We do this for each class and run each classifier on a new sample, choosing the best match from each of these. This process is performed automatically in most SVM implementations.

We saw two parameters in our previous code: **C** and kernel. We will cover the kernel parameter in the next section, but the **C** parameter is an important parameter for fitting SVMs. The **C** parameter relates to how much the classifier should aim to predict all training samples correctly, at the risk of overfitting. Selecting a higher **C** value will find a line of separation with a smaller margin, aiming to classify all training samples correctly. Choosing a lower **C** value will result in a line of separation with a larger margin—even if that means that some training samples are incorrectly classified. In this case, a lower **C** value presents a lower chance of overfitting, at the risk of choosing a generally poorer line of separation

One limitation with SVMs (in their basic form) is that they only separate data that is linearly separable. What happens if the data isn't? For that problem, we use kernels.

Kernels

When the data cannot be separated linearly, the trick is to embed it on to a higher dimensional space. What this means, with a lot of hand-waving about the details, is to add new features to the dataset until the data is linearly separable. If you add the right kinds of features, this linear separation will always, eventually, happen.

The trick is that we often compute the inner-produce of the samples when finding the best line to separate the dataset. Given a function that uses the dot product, we effectively manufacture new features without having to actually define those new features. This is known as the kernel trick and is handy because we don't know what those features were going to be anyway. We now define a kernel as a function that itself is the dot product of the function of two samples from the dataset, rather than based on the samples (and the made-up features) themselves.

We can now compute what that dot product is (or approximate it) and then just use that.

There are a number of kernels in common use. The **linear kernel** is the most straightforward and is simply the dot product of the two sample feature vectors, the weight feature, and a bias value. There is also a **polynomial kernel**, which raises the dot product to a given degree (for instance, 2). Others include the **Gaussian** (**rbf**) and **Sigmoidal** functions. In our previous code sample, we tested between the **linear** kernel and the **rbf** kernel options.

The end result from all this derivation is that these kernels effectively define a distance between two samples that is used in the classification of new samples in SVMs. In theory, any distance could be used, although it may not share the same characteristics that enable easy optimization of the SVM training.

In scikit-learn's implementation of SVMs, we can define the kernel parameter to change which kernel function is used in computations, as we saw in the previous code sample.

Character n-grams

We saw how function words can be used as features to predict the author of a document. Another feature type is character n-grams. An n-gram is a sequence of *n* tokens, where *n* is a value (for text, generally between 2 and 6). Word n-grams have been used in many studies, usually relating to the topic of the documents - as per the previous chapter. However, character n-grams have proven to be of high quality for authorship attribution.

Character n-grams are found in text documents by representing the document as a sequence of characters. These n-grams are then extracted from this sequence and a model is trained. There are a number of different models for this, but a standard one is very similar to the bag-of-words model we have used earlier.

For each distinct n-gram in the training corpus, we create a feature for it. An example of an n-gram is `<e t>`, which is the letter e, space, and then the letter t (the angle brackets are used to denote the start and end of the n-gram and are not part of the n-gram itself). We then train our model using the frequency of each n-gram in the training documents and train the classifier using the created feature matrix.

Character n-grams are defined in many ways. For instance, some applications only choose within-word characters, ignoring whitespace and punctuation. Some use this information (like our implementation in this chapter) for classification. Ultimately, this is the purpose of the model, chosen by the data miner (you!).

A common theory for why character n-grams work is that people more typically write words they can easily say and character n-grams (at least when n is between 2 and 6) are a good approximation for **phonemes**—the sounds we make when saying words. In this sense, using character n-grams approximates the sounds of words, which approximates your writing style. This is a common pattern when creating new features. First, we have a theory on what concepts will impact the end result (authorship style) and then create features to approximate or measure those concepts.

One key feature of a character n-gram matrix is that it is sparse and increases in sparsity with higher n-values quite quickly. For an n-value of 2, approximately 75 percent of our feature matrix is zeros. For an n-value of 5, over 93 percent is zeros. This is typically less sparse than a word n-gram matrix of the same type though and shouldn't cause many issues using a classifier that is used for word-based classifications.

Extracting character n-grams

We are going to use our `CountVectorizer` class to extract character n-grams. To do that, we set the analyzer parameter and specify a value for n to extract n-grams with.

The implementation in scikit-learn uses an n-gram range, allowing you to extract n-grams of multiple sizes at the same time. We won't delve into different n-values in this experiment, so we just set the values the same. To extract n-grams of size 3, you need to specify (3, 3) as the value for the n-gram range.

We can reuse the grid search from our previous code. All we need to do is specify the new feature extractor in a new pipeline and run it:

```
pipeline = Pipeline([('feature_extraction',
CountVectorizer(analyzer='char', ngram_range=(3,3))),
                     ('classifier', grid) ]
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

There is a lot of implicit overlap between function words and character n-grams, as character sequences in function words are more likely to appear. However, the actual features are very different and character n-grams capture punctuation, a characteristic that function words do not capture. For example, a character n-gram includes the full stop at the end of a sentence, while a function word-based method would only use the preceding word itself.

The Enron dataset

Enron was one of the largest energy companies in the world in the late 1990s, reporting revenue over $100 billion. It had over 20,000 staff and—as of the year 2000—there seemed to be no indications that something was very wrong.

In 2001, the *Enron Scandal* occurred, where it was discovered that Enron was undertaking systematic, fraudulent accounting practices. This fraud was deliberate, wide-ranging across the company, and for significant amounts of money. After this was publicly discovered, its share price dropped from more than $90 in 2000 to less than $1 in 2001. Enron shortly filed for bankruptcy in a mess that would take more than 5 years to finally be resolved.

As part of the investigation into Enron, the Federal Energy Regulatory Commission in the United States made more than 600,000 e-mails publicly available. Since then, this dataset has been used for research into everything from social network analysis to fraud analysis. It is also a great dataset for authorship analysis, as we are able to extract e-mails from the sent folder of individual users. This allows us to create a dataset much larger than many previous datasets.

Accessing the Enron dataset

The full set of Enron emails is available at `https://www.cs.cmu.edu/~./enron/`

The full dataset is quite large, and provided in a compression format called gzip. If you don't have a Linux-based machine to decompress (unzip) this file, get an alternative program, such as 7-zip (`http://www.7-zip.org/`)

Download the full corpus and decompress it into your data folder. By default, this will decompress into a folder called `enron_mail_20110402` which then contains a folder called `maildir`. In the Notebook, setup the data folder for the Enron dataset:

```
enron_data_folder = os.path.join(os.path.expanduser("~"), "Data",
"enron_mail_20150507", "maildir")
```

Creating a dataset loader

As we are looking for authorship information, we only want the e-mails we can attribute to a specific author. For that reason, we will look in each user's sent folder—that is, emails they have sent. We can now create a function that will choose a couple of authors at random and return each of the emails in their sent folder. Specifically, we are looking for the payloads—that is, the content rather than the e-mails themselves. For that, we will need an e-mail parser. The code is as follows:

```
from email.parser
import Parser p = Parser()
```

We will be using this later to extract the payloads from the e-mail files that are in the data folder.

With our data loading function, we are going to have a lot of options. Most of these ensure that our dataset is relatively balanced. Some authors will have thousands of e-mails in their sent mail, while others will have only a few dozen. We limit our search to only authors with at least 10 e-mails using `min_docs_author` and take a maximum of 100 e-mails from each author using the `max_docs_author` parameter. We also specify how many authors we want to get—10 by default using the `num_authors` parameter.

The function is below. Its main purpose is to loop through the authors, retrieve a number of emails for that author, and store the **document** and **class** information in some lists. We also store the mapping between an author's name and their numerical class value, which lets us retrieve that information later.

```
from sklearn.utils import check_random_state

def get_enron_corpus(num_authors=10, data_folder=enron_data_folder,
min_docs_author=10,
                     max_docs_author=100, random_state=None):
    random_state = check_random_state(random_state)
    email_addresses = sorted(os.listdir(data_folder))
    # Randomly shuffle the authors. We use random_state here to get a
repeatable shuffle
    random_state.shuffle(email_addresses)
    # Setup structures for storing information, including author
information
    documents = []
    classes = []
    author_num = 0
    authors = {}  # Maps author numbers to author names
    for user in email_addresses:
        users_email_folder = os.path.join(data_folder, user)
        mail_folders = [os.path.join(users_email_folder, subfolder)
                        for subfolder in os.listdir(users_email_folder)
                        if "sent" in subfolder]
        try:
            authored_emails = [open(os.path.join(mail_folder,
email_filename),
                                    encoding='cp1252').read()
                               for mail_folder in mail_folders
                               for email_filename in
os.listdir(mail_folder)]
        except IsADirectoryError:
            continue
        if len(authored_emails) < min_docs_author:
            continue
        if len(authored_emails) > max_docs_author:
            authored_emails = authored_emails[:max_docs_author]
        # Parse emails, store the content in documents and add to the
classes list
        contents = [p.parsestr(email)._payload for email in
authored_emails]
        documents.extend(contents)
        classes.extend([author_num] * len(authored_emails))
        authors[user] = author_num
        author_num += 1
        if author_num >= num_authors or author_num >= len(email_addresses):
```

```
            break
    return documents, np.array(classes), authors
```

It may seem odd that we sort the e-mail addresses, only to shuffle them around. The `os.listdir` function doesn't always return the same results, so we sort it first to get some stability. We then shuffle using a random state, which means our shuffling can reproduce a past result if needed.

Outside of this function, we can now get a dataset by making the following function call. We are going to use a random state of 14 here (as always in this book), but you can try other values or set it to none to get a random set each time the function is called:

```
documents, classes, authors =
get_enron_corpus(data_folder=enron_data_folder, random_state=14)
```

If you have a look at the dataset, there is still a further preprocessing set we need to undertake. Our e-mails are quite messy, but one of the worst bits (from an authorship analysis perspective) is that these e-mails contain writings from other authors, in the form of attached replies. Take the following email, which is `documents[100]`, for instance:

I would like to be on the panel but I have on a conflict on the conference

dates. Please keep me in mind for next year.

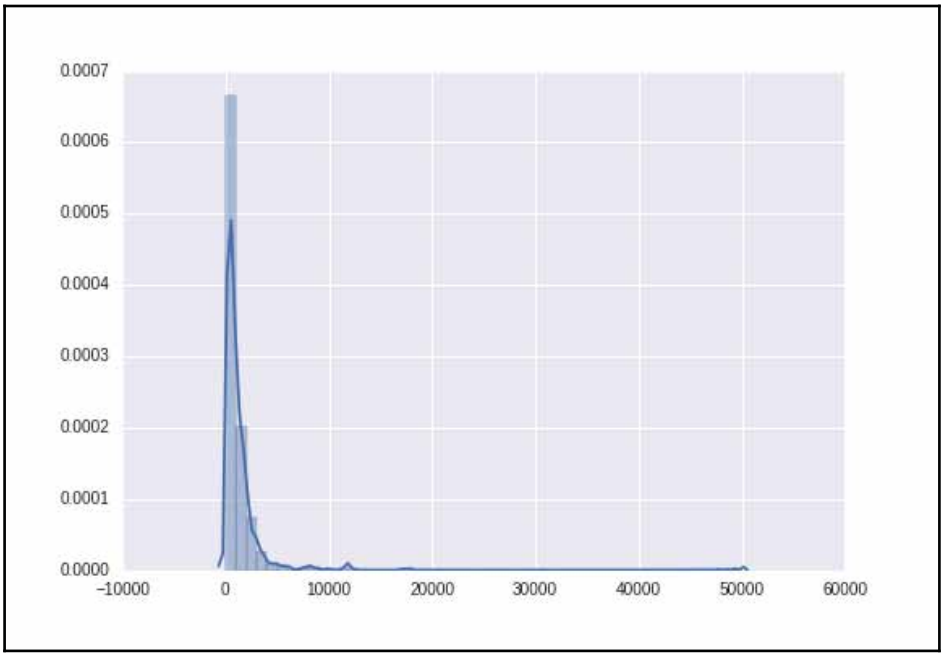
Mark Haedicke

Email is a notoriously messy format. Reply quoting, for instance, is sometimes (but not always) prepended with a > character. Other times, the reply is embedded in the original message. If you are doing larger scale data mining with email, be sure to spend more time cleaning the data to get better results.

As with the books dataset, we can plot the histogram of document lengths to get a sense of the document length distributions:

```
document_lengths = [len(document) for document in documents]
sns.distplot(document_lengths)
```

The result appears to show a strong grouping around shorter documents. While this is true, it also shows that some documents are very, very long. This may skew the results, particularly if some authors are prone to writing long documents. To compensate for this, one extension to this work may be to normalise document lengths to the first 500 characters before doing the training.



Putting it all together

We can use the existing parameter space and the existing classifier from our previous experiments—all we need to do is refit it on our new data. By default, training in scikit-learn is done from scratch—subsequent calls to `fit()` will discard any previous information.

There is a class of algorithms called *online learning* that update the training with new samples and don't restart their training each time.

As before, we can compute our scores by using `cross_val_score` and print the results. The code is as follows:

```
scores = cross_val_score(pipeline, documents, classes, scoring='f1')

print("Score: {:.3f}".format(np.mean(scores)))
```

The result is 0.683, which is a reasonable result for such a messy dataset. Adding more data (such as increasing `max_docs_author` in the dataset loading) can improve these results, as will improving the quality of the data with extra cleaning.

Evaluation

It is generally never a good idea to base an assessment on a single number. In the case of the f-score, it is usually more robust to *tricks* that give good scores despite not being useful. An example of this is accuracy. As we said in our previous chapter, a spam classifier could predict everything as being spam and get over 80 percent accuracy, although that solution is not useful at all. For that reason, it is usually worth going more in-depth on the results.

To start with, we will look at the confusion matrix, as we did in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*. Before we can do that, we need to predict a testing set. The previous code uses `cross_val_score`, which doesn't actually give us a trained model we can use. So, we will need to refit one. To do that, we need training and testing subsets:

```
from sklearn.cross_validation import train_test_split training_documents,

testing_documents, y_train, y_test = train_test_split(documents, classes,
random_state=14)
```

Next, we fit the pipeline to our training documents and create our predictions for the testing set:

```
pipeline.fit(training_documents, y_train)
y_pred = pipeline.predict(testing_documents)
```

At this point, you might be wondering what the best combination of parameters actually was. We can extract this quite easily from our grid search object (which is the classifier step of our pipeline):

```
print(pipeline.named_steps['classifier'].best_params_)
```

The results give you all of the parameters for the classifier. However, most of the parameters are the defaults that we didn't touch. The ones we did search for were C and kernel, which were set to 1 and linear, respectively.

Now we can create a confusion matrix:

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_pred, y_test)
cm = cm / cm.astype(np.float).sum(axis=1)
```

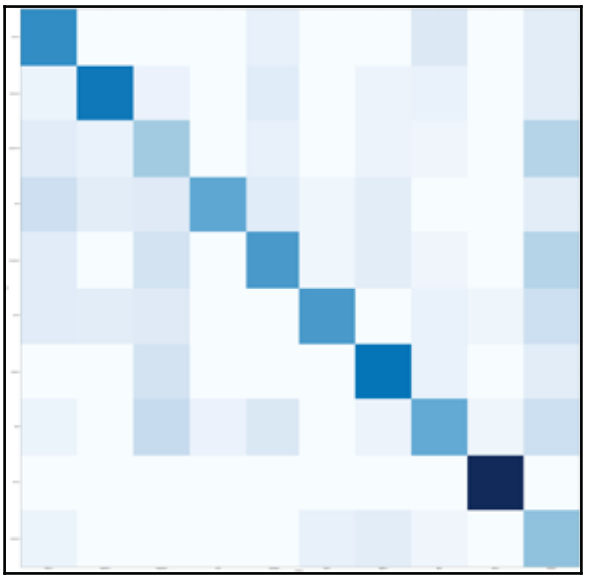
Next, we get our author's names, allowing us to that we can label the axis correctly. For this purpose, we use the authors dictionary that our Enron dataset loaded. The code is as follows:

```
sorted_authors = sorted(authors.keys(), key=lambda x:authors[x])
```

Finally, we show the confusion matrix using matplotlib. The only changes from the last chapter are highlighted below; just replace the letter labels with the authors from this chapter's experiments:

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.figure(figsize=(10,10))
plt.imshow(cm, cmap='Blues', interpolation='nearest')
tick_marks = np.arange(len(sorted_authors))
plt.xticks(tick_marks, sorted_authors)
plt.yticks(tick_marks, sorted_authors)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

The results are shown in the following figure:



We can see that authors are predicted correctly in most cases—there is a clear diagonal line with high values. There are some large sources of error though (darker values are larger): emails from user **rapp-b** are typically predicted as being from **reitmeyer-j** for instance.

Summary

In this chapter, we looked at the text mining-based problem of authorship attribution. To perform this, we analyzed two types of features: function words and character n-grams. For function words, we were able to use the bag-of-words model—simply restricted to a set of words we chose beforehand. This gave us the frequencies of only those words. For character n-grams, we used a very similar workflow using the same class. However, we changed the analyzer to look at characters and not words. In addition, we used n-grams that are sequences of n tokens in a row—in our case characters. Word n-grams are also worth testing in some applications, as they can provide a cheap way to get the context of how a word is used.

For classification, we used SVMs that optimize a line of separation between the classes based on the idea of finding the maximum margin. Anything above the line is one class and anything below the line is another class. As with the other classification tasks we have considered, we have a set of samples (in this case, our documents).

We then used a very messy dataset, the Enron e-mails. This dataset contains lots of artifacts and other issues. This resulted in a lower accuracy than the books dataset, which was much cleaner. However, we were able to choose the correct author more than half the time, out of 10 possible authors.

To take the concepts in this chapter further, look for new datasets containing authorship information. For instance, can you predict the author of a blog post? What about the author of a tweet (you may be able to reuse your data from `Chapter 6`, *Social Media Insight Using Naive Bayes*)?

In the next chapter, we consider what we can do if we don't have target classes. This is called unsupervised learning, an exploratory problem rather than a prediction problem. We also continue to deal with messy text-based datasets.

10
Clustering News Articles

In most of the earlier chapters, we performed data mining knowing what we were looking for. Our use of *target classes* allowed us to learn how our features model those targets during the training phase, which lets the algorithm set internal parameters to maximize its learning. This type of learning, where we have targets to train against, is called **supervised learning**. In this chapter, we'll consider what we do without those targets. This is **unsupervised learning** and it's much more of an exploratory task. Rather than wanting to classify with our model, the goal in unsupervised learning is to explore the data to find insights.

In this chapter, we will look at clustering news articles to find trends and patterns in the data. We'll look at how we can extract data from different websites using a link aggregation website to show a variety of news stories.

The key concepts covered in this chapter include:

- Using the reddit API to collect interesting news stories
- Obtaining text from arbitrary websites
- Cluster analysis for unsupervised data mining
- Extracting topics from documents
- Online learning for updating a model without retraining it
- Cluster ensembling to combine different models

Trending topic discovery

In this chapter, we will build a system that takes a live feed of news articles and groups them together such that the groups have similar topics. You could run the system multiple times over several weeks (or longer) to see how trends change over that time.

Our system will start with the popular link aggregation website (`https://www.reddit.com`), which stores lists of links to other websites, as well as a comments section for discussion. Links on reddit are broken into several categories of links, called **subreddits**. There are subreddits devoted to particular TV shows, funny images, and many other things. What we are interested in are the subreddits for news. We will use the */r/worldnews* subreddit in this chapter, but the code should work with any other text-based subreddit.

In this chapter, our goal is to download popular stories and then cluster them to see any major themes or concepts that occur. This will give us an insight into the popular focus, without having to manually analyze hundreds of individual stories. The general process is to:

1. Collect links to recent popular news stories from reddit.
2. Download the web page from those links.
3. Extract just the news story from the downloaded website.
4. Perform cluster analysis to find clusters of stories.
5. Analyse those clusters to discover trends.

Using a web API to get data

We have used web-based APIs to extract data in several of our previous chapters. For instance, in `Chapter 7`, *Follow Recommendations Using Graph Mining*, we used Twitter's API to extract data. Collecting data is a critical part of the data mining pipeline, and web-based APIs are a fantastic way to collect data on a variety of topics.

There are three things you need to consider when using a web-based API for collecting data: authorization methods, rate limiting, and API endpoints.

Authorization methods allow the data provider to know who is collecting the data, in order to ensure that they are being appropriately rate-limited and that data access can be tracked. For most websites, a personal account is often enough to start collecting data, but some websites will ask you to create a formal developer account to get this access.

Rate limiting is applied to data collection, particularly free services. It is important to be aware of the rules when using APIs, as they can and do change from website to website. Twitter's API limit is 180 requests per 15 minutes (depending on the particular API call). Reddit, as we will see later, allows 30 requests per minute. Other websites impose daily limits, while others limit on a per-second basis. Even within websites, there are drastic differences for different API calls. For example, Google Maps has smaller limits and different API limits per-resource, with different allowances for the number of requests per hour.

If you find you are creating an app or running an experiment that needs more requests and faster responses, most API providers have commercial plans that allow for more calls. Contact the provider for more details.

API Endpoints are the actual URLs that you use to extract information. These vary from website to website. Most often, web-based APIs will follow a RESTful interface (short for **Representational State Transfer**). RESTful interfaces often use the same actions that HTTP does: `GET`, `POST`, and `DELETE` are the most common. For instance, to retrieve information on a resource, we might use the following (example only) API endpoint:
`www.dataprovider.com/api/resource_type/resource_id/`

To get the information, we just send an HTTP `GET` request to this URL. This will return information on the resource with the given type and ID. Most APIs follow this structure, although there are some differences in the implementation. Most websites with APIs will have them appropriately documented, giving you details of all the APIs that you can retrieve.

First, we set up the parameters to connect to the service. To do this, you will need a developer key for reddit. In order to get this key, log into the site at `https://www.reddit.com/login` and go to `https://www.reddit.com/prefs/apps`. From here, click on **are you a developer? create an app...** and fill out the form, setting the type as script. You will get your client ID and a secret, which you can add to a new Jupyter Notebook:

```
CLIENT_ID = "<Enter your Client ID here>"
CLIENT_SECRET = "<Enter your Client Secret here>"
```

Reddit also asks you (when you use their API) to set the user agent to a unique string that includes your username. Create a user agent string that uniquely identifies your application. I used the name of the book, chapter 10, and a version number of 0.1 to create my user agent, but it can be any string you like. Note that not doing this may result in your connection being heavily rate-limited:

```
USER_AGENT = "python:<your unique user agent> (by /u/<your reddit
username>)"
```

In addition, you will need to log in to reddit using your username and password. If you don't have one already, sign up for a new one (it is free and you don't need to verify with personal information either).

You will need your password to complete the next step, so be careful before sharing your code to others to remove it. If you don't put your password in, set it to none and you will be prompted to enter it.

Now let's create the username and password:

```
from getpass import getpass
USERNAME = "<your reddit username>"
PASSWORD = getpass("Enter your reddit password:")
```

Next, we are going to create a function to log with this information. The reddit login API will return a token that you can use for further connections, which will be the result of this function. The code obtains the necessary information to log in to reddit, set the user agent, and then obtain an access token that we can use with future requests:

```
import requests
def login(username, password):
    if password is None:
        password = getpass.getpass("Enter reddit password for user {}:
".format(username))
    headers = {"User-Agent": USER_AGENT}
    # Setup an auth object with our credentials
    client_auth = requests.auth.HTTPBasicAuth(CLIENT_ID, CLIENT_SECRET)
    # Make a post request to the access_token endpoint
    post_data = {"grant_type": "password", "username": username,
"password": password}
    response = requests.post("https://www.reddit.com/api/v1/access_token",
auth=client_auth,
                             data=post_data, headers=headers)
    return response.json()
```

We can call now our function to get an access token:

```
token = login(USERNAME, PASSWORD)
```

This token object is just a dictionary, but it contains the `access_token` string that we will pass along with future requests. It also contains other information such as the scope of the token (which would be everything) and the time in which it expires, for example:

```
{'access_token': '<semi-random string>', 'expires_in': 3600, 'scope': '*',
'token_type': 'bearer'}
```

If you are creating a production-level app, make sure you check the expiry of the token and to refresh it if it runs out. You'll also know this has happened if your access token stops working when trying to make an API call.

Reddit as a data source

Reddit is a link aggregation website used by millions worldwide, although the English versions are US-centric. Any user can contribute a link to a website they found interesting, along with a title for that link. Other users can then upvote it, indicating that they liked the link, or downvote it, indicating they didn't like the link. The highest voted links are moved to the top of the page, while the lower ones are not shown. Older links are removed from the front page over time, depending on how many upvotes it has. Users who have stories upvoted earn points called karma, providing an incentive to submit only good stories.

Reddit also allows non-link content, called self-posts. These contain a title and some text that the submitter enters. These are used for asking questions and starting discussions. For this chapter, we will be considering only link-based posts, and not comment-based posts.

Posts are separated into different sections of the website called subreddits. A subreddit is a collection of posts that are related. When a user submits a link to reddit, they choose which subreddit it goes into. Subreddits have their own administrators, and have their own rules about what is valid content for that subreddit.

By default, posts are sorted by **Hot**, which is a function of the age of a post, the number of upvotes, the number of downvotes it has received and how liberal the content is. There is also **New**, which just gives you the most recently posted stories (and therefore contains lots of spam and bad posts), and **Top**, which is the highest voted stories for a given time period. In this chapter, we will be using Hot, which will give us recent, higher-quality stories (there really are a lot of poor-quality links in New).

Using the token we previously created, we can now obtain sets of links from a subreddit. To do that, we will use the /r/<subredditname> API endpoint that, by default, returns the Hot stories. We will use the /r/worldnews subreddit:

```
subreddit = "worldnews"
```

The URL for the previous endpoint lets us create the full URL, which we can set using string formatting:

```
url = "https://oauth.reddit.com/r/{}".format(subreddit)
```

Next, we need to set the headers. This is needed for two reasons: to allow us to use the authorization token we received earlier and to set the user agent to stop our requests from being heavily restricted. The code is as follows:

```
headers = {"Authorization": "bearer {}".format(token['access_token']),
"User-Agent": USER_AGENT}
```

Then, as before, we use the requests library to make the call, ensuring that we set the headers:

```
response = requests.get(url, headers=headers)
```

Calling `json()` on this will result in a Python dictionary containing the information returned by reddit. It will contain the top 25 results from the given subreddit. We can get the title by iterating over the stories in this response. The stories themselves are stored under the dictionary's data key. The code is as follows:

```
result = response.json()
for story in result['data']['children']:
    print(story['data']['title'])
```

Getting the data

Our dataset is going to consist of posts from the Hot list of the /r/worldnews subreddit. We saw in the previous section how to connect to reddit and how to download links. To put it all together, we will create a function that will extract the titles, links, and score for each item in a given subreddit.

We will iterate through the subreddit, getting a maximum of 100 stories at a time. We can also do pagination to get more results. We can read a large number of pages before reddit will stop us, but we will limit it to 5 pages.

As our code will be making repeated calls to an API, it is important to remember to rate-limit our calls. To do so, we will need the sleep function:

```
from time import sleep
```

Our function will accept a subreddit name and an authorization token. We will also accept a number of pages to read, though we will set a default of 5:

```
def get_links(subreddit, token, n_pages=5):
    stories = []
    after = None
    for page_number in range(n_pages):
        # Sleep before making calls to avoid going over the API limit
        sleep(2)
        # Setup headers and make call, just like in the login function
        headers = {"Authorization": "bearer
{}".format(token['access_token']), "User-Agent": USER_AGENT}
        url = "https://oauth.reddit.com/r/{}?limit=100". format(subreddit)
        if after:
            # Append cursor for next page, if we have one
            url += "&after={}".format(after)
        response = requests.get(url, headers=headers)
        result = response.json()
        # Get the new cursor for the next loop
        after = result['data']['after']
        # Add all of the news items to our stories list
        for story in result['data']['children']:
            stories.append((story['data']['title'], story['data']['url'],
story['data']['score']))
    return stories
```

We saw in `Chapter 7`, *Follow Recommendations Using Graph Mining*, how pagination works for the Twitter API. We get a cursor with our returned results, which we send with our request. Twitter will then use this cursor to get the next page of results. The reddit API does almost exactly the same thing, except it calls the parameter **after**. We don't need it for the first page, so we initially set it to **None**. We will set it to a meaningful value after our first page of results.

Calling the stories function is a simple case of passing the authorization token and the subreddit name:

```
stories = get_links("worldnews", token)
```

The returned results should contain the title, URL, and 500 stories, which we will now use to extract the actual text from the resulting websites. Here is a sample of the titles that I received by running the script:

Russia considers banning sale of cigarettes to anyone born after 2015
Swiss Muslim girls must swim with boys
Report: Russia spread fake news and disinformation in Sweden - Russia has coordinated a campaign over the past 2years to influence Sweden's decision making by using disinformation, propaganda and false documents, according to a report by researchers at The Swedish Institute of International Affairs.
100% of Dutch Trains Now Run on Wind Energy. The Netherlands met its renewable energy goals a year ahead of time.
Legal challenge against UK's sweeping surveillance laws quickly crowdfunded
A 1,000-foot-thick ice block about the size of Delaware is snapping off of Antarctica
The U.S. dropped an average of 72 bombs every day — the equivalent of three an hour — in 2016, according to an analysis of American strikes around the world. U.S. Bombed Iraq, Syria, Pakistan, Afghanistan, Libya, Yemen, Somalia in 2016
The German government is investigating a recent surge in fake news following claims that Russia is attempting to meddle in the country's parliamentary elections later this year.
Pesticides kill over 10 million bees in a matter of days in Brazil countryside
The families of American victims of Islamic State terrorist attacks in Europe have sued Twitter, charging that the social media giant allowed the terror group to proliferate online
Gas taxes drop globally despite climate change; oil & gas industry gets $500 billion in subsidies; last new US gas tax was in 1993
Czech government tells citizens to arm themselves and shoot Muslim terrorists in case of 'Super Holocaust'
PLO threatens to revoke recognition of Israel if US embassy moves to Jerusalem
Two-thirds of all new HIV cases in Europe are being recorded in just one country – Russia: More than a million Russians now live with the virus and that number is expected to nearly double in the next decade
Czech government tells its citizens how to fight terrorists: Shoot them yourselves | The interior ministry is pushing a constitutional change that would let citizens use guns against terrorists
Morocco Prohibits Sale of Burqa
Mass killer Breivik makes Nazi salute at rights appeal case
Soros Groups Risk Purge After Trump's Win Emboldens Hungary
Nigeria purges 50,000 'ghost workers' from State payroll in corruption sweep
Alcohol advertising is aggressive and linked to youth drinking, research finds | Society
UK Government quietly launched 'assault on freedom' while distracting people, say campaigners behind legal challenge - The Investigatory Powers Act became law at the end of last year, and gives spies the power to read through everyone's entire internet history
Russia's Reserve Fund down 70 percent in 2016

Russian diplomat found dead in Athens
At least 21 people have been killed (most were civilians) and 45 wounded in twin bombings near the Afghan parliament in Kabul
Pound's Decline Deepens as Currency Reclaims Dubious Honor

World news isn't usually the most optimistic of places, but it does give insight into what is going on around the world, and trends on this subreddit are usually indicative of trends in the world.

Extracting text from arbitrary websites

The links that we get from reddit go to arbitrary websites run by many different organizations. To make it harder, those pages were designed to be read by a human, not a computer program. This can cause a problem when trying to get the actual content/story of those results, as modern websites have a lot going on in the background. JavaScript libraries are called, style sheets are applied, advertisements are loaded using AJAX, extra content is added to sidebars, and various other things are done to make the modern web page a complex document. These features make the modern Web what it is, but make it difficult to automatically get good information from!

Finding the stories in arbitrary websites

To start with, we will download the full web page from each of these links and store them in our data folder, under a raw subfolder. We will process these to extract the useful information later on. This caching of results ensures that we don't have to continuously download the websites while we are working. First, we set up the data folder path:

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "websites",
"raw")
```

We are going to use MD5 hashing to create unique filenames for our articles, by hashing the URL, and we will import `hashlib` to do that. A `hash` function is a function that converts some input (in our case a string containing the title) into a string that is seemingly random. The same input will always return the same output, but slightly different inputs will return drastically different outputs. It is also impossible to go from a hash value to the original value, making it a one-way function.

```
import hashlib
```

For this chapter's experiments, we are going to simply skip any website downloads that fail. In order to make sure we don't lose too much information doing this, we maintain a simple counter of the number of errors that occur. We are going to suppress any error that occurs, which could result in a systematic problem prohibiting downloads. If this error counter is too high, we can look at what those errors were and try to fix them. For example, if the computer has no internet access, all 500 of the downloads will fail and you should probably fix that before continuing!

```
number_errors = 0
```

Next, we iterate through each of our stories, download the website, and save the results to a file:

```
for title, url, score in stories:
    output_filename = hashlib.md5(url.encode()).hexdigest()
    fullpath = os.path.join(data_folder, output_filename + ".txt")
    try:
        response = requests.get(url)
        data = response.text
        with open(fullpath, 'w') as outf:
            outf.write(data)
        print("Successfully completed {}".format(title))
    except Exception as e:
        number_errors += 1
        # You can use this to view the errors, if you are getting too many:
        # raise
```

If there is an error in obtaining the website, we simply skip this website and keep going. This code will work on a large number of websites and that is good enough for our application, as we are looking for general trends and not exactness.

Note that sometimes you do care about getting 100 percent of responses, and you should adjust your code to accommodate more errors. Be warned though that there is a significant increase in effort required to create code the works reliably on data from the internet. The code to get those final 5 to 10 percent of websites will be significantly more complex.

In the preceding code, we simply catch any error that happens, record the error and move on.

If you find that too many errors occur, change the print(e) line to just type raise instead. This will cause the exception to be called, allowing you to debug the problem.

After this has completed, we will have a bunch of websites in our `raw` subfolder. After taking a look at these pages (open the created files in a text editor), you can see that the content is there but there is HTML, JavaScript, CSS code, as well as other content. As we are only interested in the story itself, we now need a way to extract this information from these different websites.

Extracting the content

After we get the raw data, we need to find the story in each. There are several complex algorithms for doing this, as well as some simple ones. We will stick with a simple method here, keeping in mind that often enough, the simple algorithm is good enough. This is part of data mining—knowing when to use simple algorithms to get a job done, versus using more complicated algorithms to obtain that extra bit of performance.

First, we get a list of each of the filenames in our `raw` subfolder:

```
filenames = [os.path.join(data_folder, filename) for filename in
os.listdir(data_folder)]
```

Next, we create an output folder for the text-only versions that we will extract:

```
text_output_folder = os.path.join(os.path.expanduser("~"), "Data",
"websites", "textonly")
```

Next, we develop the code that will extract the text from the files. We will use the lxml library to parse the HTML files, as it has a good HTML parser that deals with some badly formed expressions. The code is as follows:

```
from lxml import etree
```

The actual code for extracting text is based on three steps:

1. We iterate through each of the nodes in the HTML file and extract the text in it.
2. We skip any node that is JavaScript, styling, or a comment, as this is unlikely to contain information of interest to us.
3. We ensure that the content has at least 100 characters. This is a good baseline, but it could be improved upon for more accurate results.

As we said before, we aren't interested in scripts, styles, or comments. So, we create a list to ignore nodes of those types. Any node that has a type in this list will not be considered as containing the story. The code is as follows:

```
skip_node_types = ["script", "head", "style", etree.Comment]
```

We will now create a function that parses an HTML file into an lxml `etree`, and then we will create another function that parses this tree looking for text. This first function is pretty straightforward; simply open the file and create a tree using the lxml library's parsing function for HTML files. The code is as follows:

```
parser = etree.HTMLParser()

def get_text_from_file(filename):
    with open(filename) as inf:
        html_tree = etree.parse(inf, parser)
    return get_text_from_node(html_tree.getroot())
```

In the last line of that function, we call the `getroot()` function to get the root node of the tree, rather than the full `etree`. This allows us to write our text extraction function to accept any node, and therefore write a recursive function.

This function will call itself on any child nodes to extract the text from them, and then return the concatenation of any child nodes text.

If the node where this function is passed doesn't have any child nodes, we just return the text from it. If it doesn't have any text, we just return an empty string. Note that we also check here for our third condition—that the text is at least 100 characters long.

The code for checking that the text is at least 100 characters long is as follows:

```
def get_text_from_node(node):
    if len(node) == 0:
        # No children, just return text from this item
        if node.text:
            return node.text
        else:
            return ""
    else:
        # This node has children, return the text from it:
        results = (get_text_from_node(child)
                   for child in node
                   if child.tag not in skip_node_types)
    result = str.join("n", (r for r in results if len(r) > 1))
    if len(result) >= 100:
```

```
            return result
        else:
            return ""
```

At this point, we know that the node has child nodes, so we recursively call this function on each of those child nodes and then join the results when they return.

The final condition inside the return line stops blank lines being returned (for example, when a node has no children and no text). We also use a generator, which makes the code more efficient by only grabbing text data when it is needed, namely the final return statement rather than creating a number of sub-lists.

We can now run this code on all of the raw HTML pages by iterating through them, calling the text extraction function on each, and saving the results to the text-only subfolder:

```
for filename in os.listdir(data_folder):
    text = get_text_from_file(os.path.join(data_folder, filename))
    with open(os.path.join(text_output_folder, filename), 'w') as outf:
        outf.write(text)
```

You can evaluate the results manually by opening each of the files in the text only subfolder and checking their content. If you find too many of the results have non-story content, try increasing the minimum-100-character limit. If you still can't get good results, or need better results for your application, try the methods listed in *Appendix A, Next Steps.*

Grouping news articles

The aim of this chapter is to discover trends in news articles by clustering, or grouping, them together. To do that, we will use the k-means algorithm, a classic machine learning algorithm originally developed in 1957.

Clustering is an unsupervised learning technique and we often use clustering algorithms for exploring data. Our dataset contains approximately 500 stories and it would be quite arduous to examine each of those stories individually. Using clustering allows us to group similar stories together, and we can explore the themes in each cluster independently.

We use clustering techniques when we don't have a clear set of target classes for our data. In that sense, clustering algorithms have little direction in their learning. They learn according to some function, regardless of the underlying meaning of the data.

For this reason, it is critical to choose good features. In supervised learning, if you choose poor features, the learning algorithm can choose to not use those features. For instance, support vector machines will give little weight to features that aren't useful in classification. However, with clustering, all features are used in the final result—even if those features don't provide us with the answer we were looking for.

When performing cluster analysis on real-world data, it is always a good idea to have a sense of what sorts of features will work for your scenario. In this chapter, we will use the bag-of-words model. We are looking for topic-based groups, so we will use topic-based features to model the documents. We know those features work because of the work others have done in supervised versions of our problem. In contrast, if we were to perform an authorship-based clustering, we would use features such as those found in the `Chapter 9`, *Authorship Attribution*, experiment.

The k-means algorithm

The k-means clustering algorithm finds centroids that best represent the data using an iterative process. The algorithm starts with a predefined set of centroids, which are normally data points taken from the training data. The **k** in k-means is the number of centroids to look for and how many clusters the algorithm will find. For instance, setting k to 3 will find three clusters in the dataset.

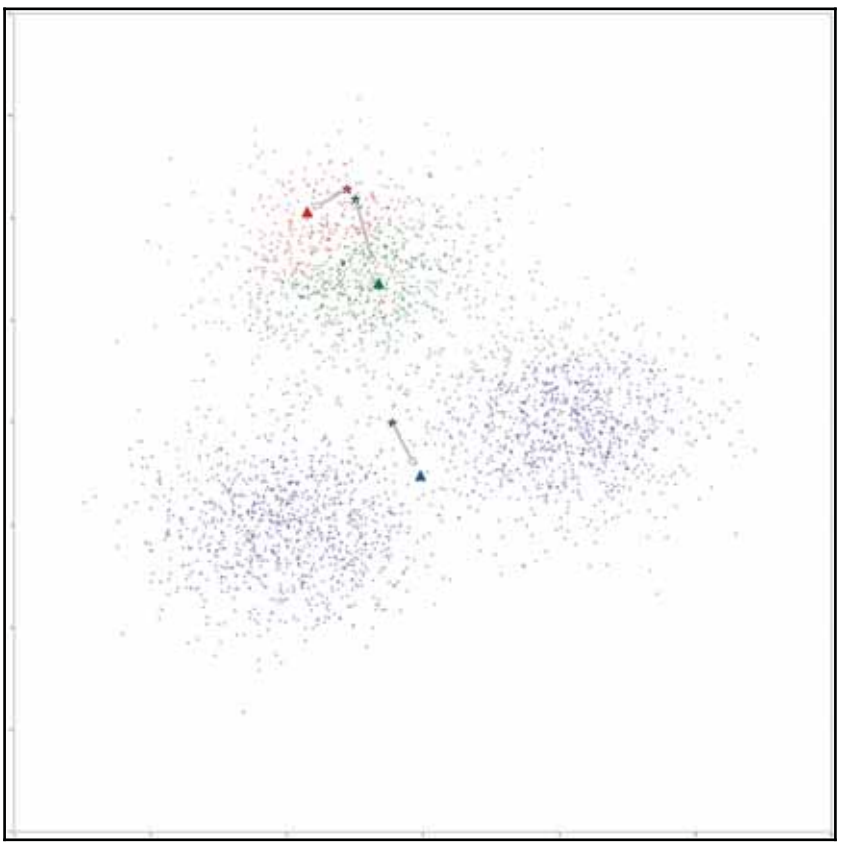
There are two phases to the k-means: **assignment** and **updating**. They are explained as below:

- In the **assignment** step, we set a label to every sample in the dataset linking it to the nearest centroid. For each sample nearest to centroid 1, we assign the label 1. For each sample nearest to centroid 2, we assign a label 2 and so on for each of the k centroids. These labels form the clusters, so we say that each data point with the label 1 is in cluster 1 (at this time only, as assignments can change as the algorithm runs).
- In the **updating** step, we take each of the clusters and compute the centroid, which is the mean of all of the samples in that cluster.

The algorithm then iterates between the assignment step and the updating step; each time the updating step occurs, each of the centroids moves a small amount. This causes the assignments to change slightly, causing the centroids to move a small amount in the next iteration. This repeats until some stopping criterion is reached.

It is common to stop after a certain number of iterations, or when the total movement of the centroids is very low. The algorithm can also complete in some scenarios, which means that the clusters are stable—the assignments do not change and neither do the centroids.

In the following figure, k-means was performed over a dataset created randomly, but with three clusters in the data. The stars represent the starting location of the centroids, which were chosen randomly by picking a random sample from the dataset. Over 5 iterations of the k-means algorithm, the centroids move to the locations represented by the triangles.



The k-means algorithm is fascinating for its mathematical properties and historical significance. It is an algorithm that (roughly) only has a single parameter, and is quite effective and frequently used, even more than 50 years after its discovery.

There is a k-means algorithm in scikit-learn, which we import from the `cluster` module in scikit-learn:

```
from sklearn.cluster import KMeans
```

We also import the `CountVectorizer` class's close cousin, `TfidfVectorizer`. This vectorizer applies a weighting to each term's counts, depending on how many documents it appears in, using the equation: **tf / log(df)**, where tf is a term's frequency (how many times it appears in the current document) and df is the term's document frequency (how many documents in our corpus it appears in). Terms that appear in many documents are weighted lower (by dividing the value by the log of the number of documents it appears in). For many text mining applications, using this type of weighting scheme can improve performance quite reliably. The code is as follows:

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

We then set up our pipeline for our analysis. This has two steps. The first is to apply our vectorizer, and the second is to apply our k-means algorithm. The code is as follows:

```
from sklearn.pipeline import Pipeline
n_clusters = 10
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_df=0.4)),
                                        ('clusterer',
KMeans(n_clusters=n_clusters)) ])
```

The `max_df` parameter is set to a low value of 0.4, which says ignore any word that occurs in more than 40 percent of documents. This parameter is invaluable for removing function words that give little topic-based meaning on their own.

Removing any word that occurs in more than 40 percent of documents will remove function words, making this type of preprocessing quite useless for the work we saw in `Chapter 9`, *Authorship Attribution*.

```
documents = [open(os.path.join(text_output_folder, filename)).read()
             for filename in os.listdir(text_output_folder)]
```

We then fit and predict this pipeline. We have followed this process a number of times in this book so far for classification tasks, but there is a difference here—we do not give the target classes for our dataset to the fit function. This is what makes this an unsupervised learning task! The code is as follows:

```
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

The labels variable now contains the cluster numbers for each sample. Samples with the same label are said to belong to the same cluster. It should be noted that the cluster labels themselves are meaningless: clusters 1 and 2 are no more similar than clusters 1 and 3.

We can see how many samples were placed in each cluster using the `Counter` class:

```
from collections import Counter
c = Counter(labels)
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

```
Cluster 0 contains 1 samples
Cluster 1 contains 2 samples
Cluster 2 contains 439 samples
Cluster 3 contains 1 samples
Cluster 4 contains 2 samples
Cluster 5 contains 3 samples
Cluster 6 contains 27 samples
Cluster 7 contains 2 samples
Cluster 8 contains 12 samples
Cluster 9 contains 1 samples
```

Many of the results (keeping in mind that your dataset will be quite different to mine) consist of a large cluster with the majority of instances, several medium clusters, and some clusters with only one or two instances. This imbalance is quite normal in many clustering applications.

Evaluating the results

Clustering is mainly an exploratory analysis, and therefore it is difficult to evaluate a clustering algorithm's results effectively. A straightforward way is to evaluate the algorithm based on the criteria the algorithm tries to learn from.

If you have a test set, you can evaluate clustering against it. For more details, visit `http://nlp.standford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`

In the case of the k-means algorithm, the criterion that it uses when developing the centroids is to minimize the distance from each sample to its nearest centroid. This is called the inertia of the algorithm and can be retrieved from any KMeans instance that has had fit called on it:

```
pipeline.named_steps['clusterer'].inertia_
```

The result on my dataset was 343.94. Unfortunately, this value is quite meaningless by itself, but we can use it to determine how many clusters we should use. In the preceding example, we set `n_clusters` to 10, but is this the best value? The following code runs the k-means algorithm 10 times with each value of `n_clusters` from 2 to 20, taking some time to complete the large number of runs.

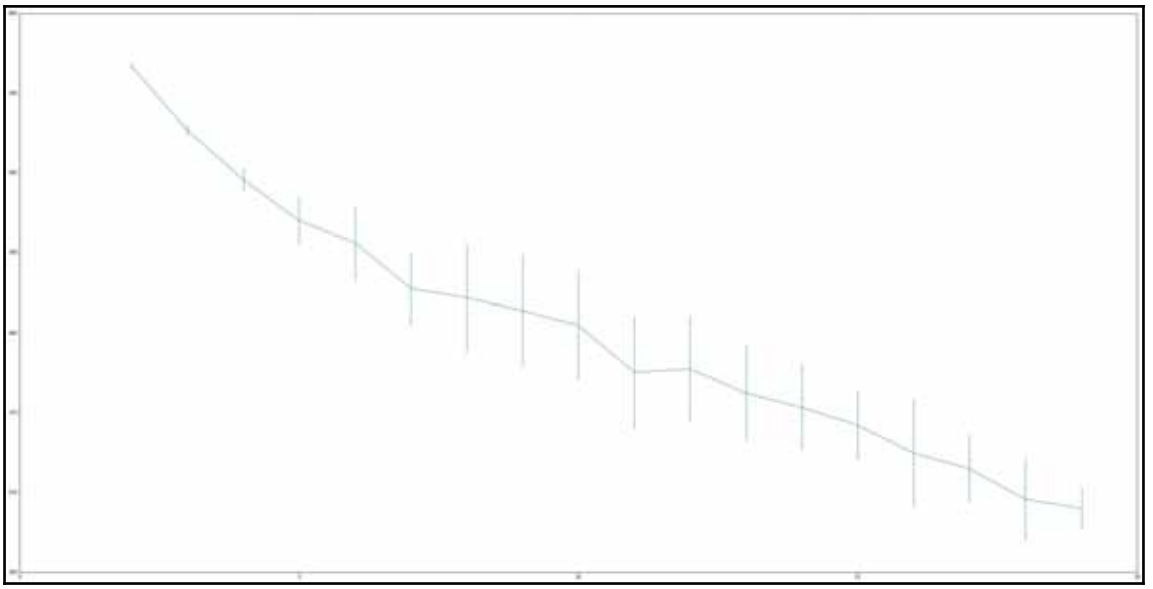
For each run, it records the inertia of the result.

You may notice the following code that we don't use a Pipeline; instead, we split out the steps. We only create the X matrix from our text documents once per value of `n_clusters` to (drastically) improve the speed of this code.

```
inertia_scores = []
n_cluster_values = list(range(2, 20))
for n_clusters in n_cluster_values:
    cur_inertia_scores = []
    X = TfidfVectorizer(max_df=0.4).fit_transform(documents)
 for i in range(10):
        km = KMeans(n_clusters=n_clusters).fit(X)
        cur_inertia_scores.append(km.inertia_)
    inertia_scores.append(cur_inertia_scores)
```

The`inertia_scores` variable now contains a list of inertia scores for each n_clusters value between 2 and 20. We can plot this to get a sense of how this value interacts with `n_clusters`:

```
%matplotlib inline
from matplotlib import pyplot as plt
inertia_means = np.mean(inertia_scores, axis=1)
inertia_stderr = np.std(inertia_scores, axis=1)
fig = plt.figure(figsize=(40,20))
plt.errorbar(n_cluster_values, inertia_means, inertia_stderr,
color='green')
plt.show()
```



Overall, the value of the inertia should decrease with reducing improvement as the number of clusters improves, which we can broadly see from these results. The increase between values of 6 to 7 is due only to the randomness in selecting the centroids, which directly affect how good the final results are. Despite this, there is a general trend (for my data; your results may vary) that about 6 clusters was the last time a major improvement in the inertia occurred.

After this point, only slight improvements are made to the inertia, although it is hard to be specific about vague criteria such as this. Looking for this type of pattern is called the elbow rule, in that we are looking for an elbow-esque bend in the graph. Some datasets have more pronounced elbows, but this feature isn't guaranteed to even appear (some graphs may be smooth!).

Based on this analysis, we set `n_clusters` to be 6 and then rerun the algorithm:

```
n_clusters = 6
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters)) ])
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

Extracting topic information from clusters

Now we set our sights on the clusters in an attempt to discover the topics in each.

We first extract the term list from our feature extraction step:

```
terms = pipeline.named_steps['feature_extraction'].get_feature_names()
```

We also set up another counter for counting the size of each of our classes:

```
c = Counter(labels)
```

Iterating over each cluster, we print the size of the cluster as before.

It is important to keep in mind the sizes of the clusters when evaluating the results—some of the clusters will only have one sample, and are therefore not indicative of a general trend.

Next (and still in the loop), we iterate over the most important terms for this cluster. To do this, we take the five largest values from the centroid, which we get by finding the features that have the highest values in the centroid itself.

```
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
    print(" Most important terms")
    centroid =
pipeline.named_steps['clusterer'].cluster_centers_[cluster_number]
    most_important = centroid.argsort()
    for i in range(5):
        term_index = most_important[-(i+1)]
        print(" {0}) {1} (score: {2:.4f})".format(i+1, terms[term_index],
centroid[term_index]))
```

The results can be quite indicative of current trends. In my results (obtained January 2017), the clusters correspond to health matters, Middle East tensions, Korean tensions, and Russian affairs. These were the main topics frequenting news around this time—although this has hardly changed for a number of years!

You might notice some words that don't provide much value come out on top, such as *you*, *her* and *mr*. These function words are great for authorship analysis - as we saw in `Chapter 9`, *Authorship Attribution*, but are not generally very good for topic analysis. Passing the list of function words into the `stop_words` parameter of the **TfidfVectorizer** in our pipeline above will ignore those words. Here is the updated code for building such a pipeline:

```
function_words = [... list from Chapter 9 ...]

pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_df=0.4,
stop_words=function_words)),
                     ('clusterer', KMeans(n_clusters=n_clusters)) ])
```

Using clustering algorithms as transformers

As a side note, one interesting property about the k-means algorithm (and any clustering algorithm) is that you can use it for feature reduction. There are many methods to reduce the number of features (or create new features to embed the dataset on), such as **Principle Component Analysis**, **Latent Semantic Indexing**, and many others. One issue with many of these algorithms is that they often need lots of computing power.

In the preceding example, the terms list had more than 14,000 entries in it—it is quite a large dataset. Our k-means algorithm transformed these into just six clusters. We can then create a dataset with a much lower number of features by taking the distance to each centroid as a feature.

To do this, we call the transform function on a KMeans instance. Our pipeline is fit for this purpose, as it has a k-means instance at the end:

```
X = pipeline.transform(documents)
```

This calls the transform method on the final step of the pipeline, which is an instance of k-means. This results in a matrix that has six features and the number of samples is the same as the length of documents.

You can then perform your own second-level clustering on the result, or use it for classification if you have the target values. A possible workflow for this would be to perform some feature selection using the supervised data, use clustering to reduce the number of features to a more manageable number, and then use the results in a classification algorithm such as SVMs.

Clustering ensembles

In `Chapter 3`, *Predicting Sports Winners with Decision Trees*, we looked at a classification ensemble using the random forest algorithm, which is an ensemble of many low-quality, tree-based classifiers. Ensembling can also be performed using clustering algorithms. One of the key reasons for doing this is to smooth the results from many runs of an algorithm. As we saw before, the results from running k-means are varied, depending on the selection of the initial centroids. Variation can be reduced by running the algorithm many times and then combining the results.

Ensembling also reduces the effects of choosing parameters on the final result. Most clustering algorithms are quite sensitive to the parameter values chosen for the algorithm. Choosing slightly different parameters results in different clusters.

Evidence accumulation

As a basic ensemble, we can first cluster the data many times and record the labels from each run. We then record how many times each pair of samples was clustered together in a new matrix. This is the essence of the **Evidence Accumulation Clustering** (**EAC**) algorithm.

EAC has two major steps.

1. The first step is to cluster the data many times using a lower-level clustering algorithm, such as k-means and record the frequency that samples were in the same cluster, in each iteration. This is stored in a **co-association matrix**.
2. The second step is to perform a cluster analysis on the resulting co-association matrix, which is performed using another type of clustering algorithm called hierarchical clustering. This has an interesting graph-theory-based property, as it is mathematically the same as finding a tree that links all the nodes together and removing weak links.

We can create a co-association matrix from an array of labels by iterating over each of the labels and recording where two samples have the same label. We use SciPy's `csr_matrix`, which is a type of sparse matrix:

```
from scipy.sparse import csr_matrix
```

Our function definition takes a set of labels and then record the rows and columns of each match. We do these in a list. Sparse matrices are commonly just sets of lists recording the positions of nonzero values, and `csr_matrix` is an example of this type of sparse matrix. For each pair of samples with the same label, we record the position of both samples in our list:

```
import numpy as np
def create_coassociation_matrix(labels):
    rows = []
    cols = []
    unique_labels = set(labels)
    for label in unique_labels:
        indices = np.where(labels == label)[0]
        for index1 in indices:
            for index2 in indices:
                rows.append(index1)
                cols.append(index2)
    data = np.ones((len(rows),))
    return csr_matrix((data, (rows, cols)), dtype='float')
```

To get the co-association matrix from the labels, we simply call this function:

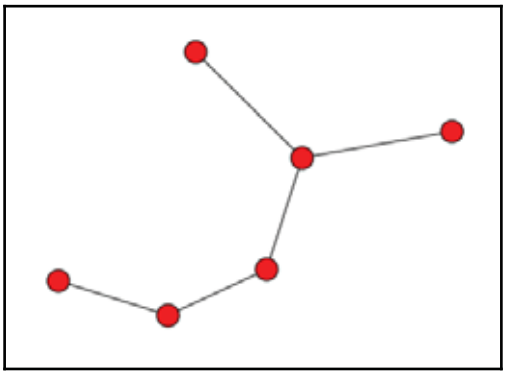
```
C = create_coassociation_matrix(labels)
```

From here, we can add multiple instances of these matrices together. This allows us to combine the results from multiple runs of k-means. Printing out `C` (just enter C into a new cell of your Jupyter Notebook and run it) will tell you how many cells have nonzero values in them. In my case, about half of the cells had values in them, as my clustering result had a large cluster (the more even the clusters, the lower the number of nonzero values).

The next step involves the hierarchical clustering of the co-association matrix. We will do this by finding minimum spanning trees on this matrix and removing edges with a weight lower than a given threshold.

In graph theory, a spanning tree is a set of edges on a graph that connects all of the nodes together. The **Minimum Spanning Tree** (MST) is simply the spanning tree with the lowest total weight. For our application, the nodes in our graph are samples from our dataset, and the edge weights are the number of times those two samples were clustered together—that is, the value from our co-association matrix.

In the following figure, a MST on a graph of six nodes is shown. Nodes on the graph can be connected to more than once in the MST, as long as all nodes are connected together.



To compute the MST, we use SciPy's `minimum_spanning_tree` function, which is found in the sparse package:

```
from scipy.sparse.csgraph import minimum_spanning_tree
```

The `mst` function can be called directly on the sparse matrix returned by our co-association function:

```
mst = minimum_spanning_tree(C)
```

However, in our co-association matrix C, higher values are indicative of samples that are clustered together more often—a similarity value. In contrast, `minimum_spanning_tree` sees the input as a distance, with higher scores penalized. For this reason, we compute the minimum spanning tree on the negation of the co-association matrix instead:

```
mst = minimum_spanning_tree(-C)
```

The result from the preceding function is a matrix the same size as the co-association matrix (the number of rows and columns is the same as the number of samples in our dataset), with only the edges in the MST kept and all others removed.

We then remove any node with a weight less than a predefined threshold. To do this, we iterate over the edges in the MST matrix, removing any that are less than a specific value. We can't test this out with just a single iteration in a co-association matrix (the values will be either 1 or 0, so there isn't much to work with). So, we will create extra labels first, create the co-association matrix, and then add the two matrices together. The code is as follows:

```
pipeline.fit(documents)
labels2 = pipeline.predict(documents)
C2 = create_coassociation_matrix(labels2)
C_sum = (C + C2) / 2
```

We then compute the MST and remove any edge that didn't occur in both of these labels:

```
mst = minimum_spanning_tree(-C_sum)
mst.data[mst.data > -1] = 0
```

The threshold we wanted to cut off was any edge not in both clusterings—that is, with a value of 1. However, as we negated the co-association matrix, we had to negate the threshold value too.

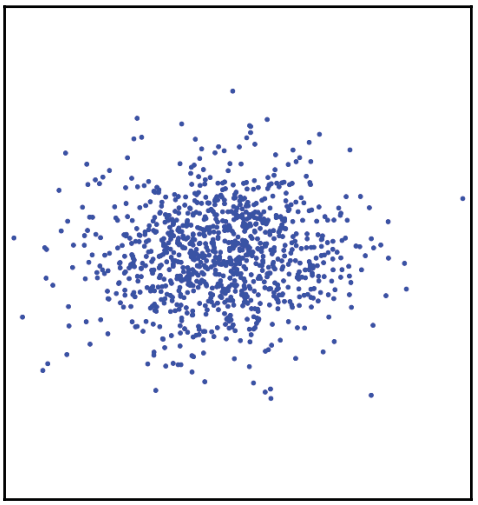
Lastly, we find all of the connected components, which is simply a way to find all of the samples that are still connected by edges after we removed the edges with low weights. The first returned value is the number of connected components (that is, the number of clusters) and the second is the labels for each sample. The code is as follows:

```
from scipy.sparse.csgraph import connected_components
number_of_clusters, labels = connected_components(mst)
```

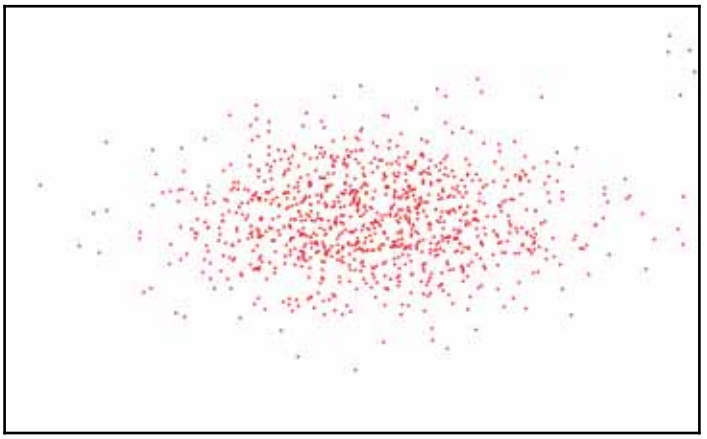
In my dataset, I obtained eight clusters, with the clusters being approximately the same as before. This is hardly a surprise, given we only used two iterations of k-means; using more iterations of k-means (as we do in the next section) will result in more variance.

How it works

In the k-means algorithm, each feature is used without any regard to its weight. In essence, all features are assumed to be on the same scale. We saw the problems with not scaling features in `Chapter 2`, *Classification with scikit-learn Estimators*. The result of this is that k-means is looking for circular clusters, visualized here:

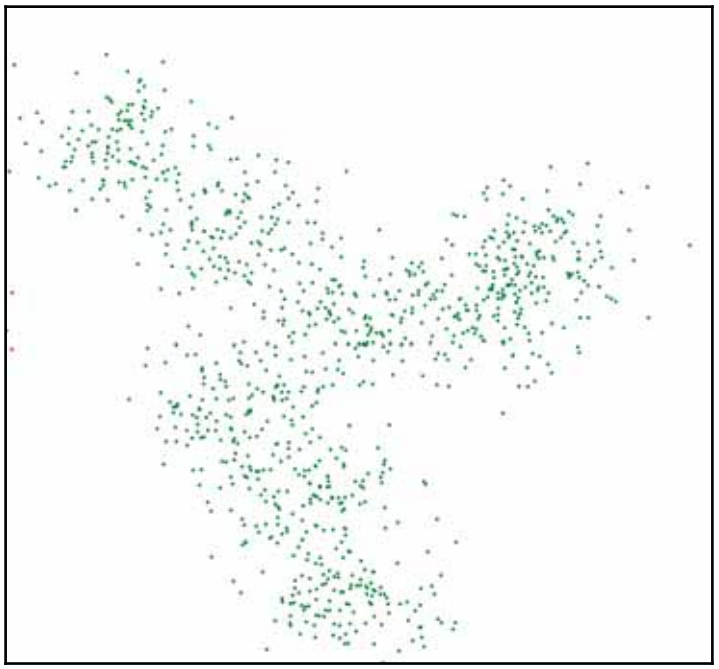


Oval shaped clusters can also be discovered by k-means. The separation usually isn't quite so smooth, but can be made easier with feature scaling. An example of this shaped cluster is as follows:



As we can see in the preceding screenshot, not all clusters have this shape. The blue cluster is circular and is of the type that k-means is very good at picking up. The red cluster is an ellipse. The k-means algorithm can pick up clusters of this shape with some feature scaling.

The bellow third cluster isn't even convex—it is an odd shape that k-means will have trouble discovering, but would still be considered a *cluster*, at least by most humans looking at the picture:



Cluster analysis is a hard task, with most of the difficulty simply in trying to define the problem. Many people intuitively understand what it means, but trying to define it in precise terms (necessary for machine learning) is very difficult. Even people often disagree on the term!

The EAC algorithm works by remapping the features onto a new space, in essence turning each run of the k-means algorithm into a transformer using the same principles we saw the previous section using k-means for feature reduction. In this case, though, we only use the actual label and not the distance to each centroid. This is the data that is recorded in the co-association matrix.

The result is that EAC now only cares about how close things are to each other, not how they are placed in the original feature space. There are still issues around unscaled features. Feature scaling is important and should be done anyway (we did it using tf-idf in this chapter, which results in feature values having the same scale).

We saw a similar type of transformation in `Chapter 9`, *Authorship Attribution*, through the use of kernels in SVMs. These transformations are very powerful and should be kept in mind for complex datasets. The algorithms for remapping data onto a new feature space does not need to be complex though, as you'll see in the EAC algorithm.

Implementation

Putting all this all together, we can now create a clustering algorithm fitting the scikit-learn interface that performs all of the steps in EAC. First, we create the basic structure of the class using scikit-learn's *ClusterMixin*.

Our parameters are the number of k-means clusterings to perform in the first step (to create the co-association matrix), the threshold to cut off at, and the number of clusters to find in each k-means clustering. We set a range of n_clusters in order to get lots of variance in our k-means iterations. Generally, in ensemble terms, variance is a good thing; without it, the solution can be no better than the individual clusterings (that said, high variance is not an indicator that the ensemble will be better).

I'll present the full class first, and then overview each of the functions:

```
from sklearn.base import BaseEstimator, ClusterMixin
class EAC(BaseEstimator, ClusterMixin):
    def __init__(self, n_clusterings=10, cut_threshold=0.5,
n_clusters_range=(3, 10)):
        self.n_clusterings = n_clusterings
        self.cut_threshold = cut_threshold
        self.n_clusters_range = n_clusters_range

    def fit(self, X, y=None):
        C = sum((create_coassociation_matrix(self._single_clustering(X))
                 for i in range(self.n_clusterings)))
        mst = minimum_spanning_tree(-C)
        mst.data[mst.data > -self.cut_threshold] = 0
        mst.eliminate_zeros()
        self.n_components, self.labels_ = connected_components(mst)
        return self

    def _single_clustering(self, X):
        n_clusters = np.random.randint(*self.n_clusters_range)
```

```
            km = KMeans(n_clusters=n_clusters)
            return km.fit_predict(X)

        def fit_predict(self, X):
            self.fit(X)
            return self.labels_
```

The goal of the `fit` function is to perform the k-means clusters a number of times, combine the co-association matrices and then split it by finding the MST, as we saw earlier with the EAC example. We then perform our low-level clustering using k-means and sum the resulting co-association matrices from each iteration. We do this in a generator to save memory, creating only the co-association matrices when we need them. In each iteration of this generator, we create a new single k-means run with our dataset and then create the co-association matrix for it. We use `sum` to add these together.

As before, we create the MST, remove any edges less than the given threshold (properly negating values as explained earlier), and find the connected components. As with any fit function in scikit-learn, we need to return self in order for the class to work in pipelines effectively.

The `_single_clustering` function is designed to perform a single iteration of k-means on our data, and then return the predicted labels. To do this, we randomly choose a number of clusters to find using NumPy's `randint` function and our `n_clusters_range` parameter, which sets the range of possible values. We then cluster and predict the dataset using k-means. The return value here will be the labels coming from k-means.

Finally, the `fit_predict` function simply calls **fit**, and then returns the labels for the documents.

We can now run this on our previous code by setting up a pipeline as before and using EAC where we previously used a KMeans instance as our final stage of the pipeline. The code is as follows:

```
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_df=0.4)),
                     ('clusterer', EAC()) ])
```

Online learning

In some cases, we don't have all of the data we need for training before we start our learning. Sometimes, we are waiting for new data to arrive, perhaps the data we have is too large to fit into memory, or we receive extra data after a prediction has been made. In cases like these, online learning is an option for training models over time.

Online learning is the incremental updating of a model as new data arrives. Algorithms that support online learning can be trained on one or a few samples at a time, and updated as new samples arrive. In contrast, algorithms that are not **online** require access to all of the data at once. The standard k-means algorithm is like this, as are most of the algorithms we have seen so far in this book.

Online versions of algorithms have a means to partially update their model with only a few samples. Neural networks are a standard example of an algorithm that works in an online fashion. As a new sample is given to the neural network, the weights in the network are updated according to a learning rate, which is often a very small value such as 0.01. This means that any single instance only makes a small (but hopefully improving) change to the model.

Neural networks can also be trained in batch mode, where a group of samples is given at once and the training is done in one step. Algorithms are generally faster in batch mode but use more memory.

In this same vein, we can slightly update the k-means centroids after a single or small batch of samples. To do this, we apply a learning rate to the centroid movement in the updating step of the k-means algorithm. Assuming that samples are randomly chosen from the population, the centroids should tend to move towards the positions they would have in the standard, offline, and k-means algorithm.

Online learning is related to streaming-based learning; however, there are some important differences. Online learning is capable of reviewing older samples after they have been used in the model, while a streaming-based machine learning algorithm typically only gets one pass—that is, one opportunity to look at each sample.

Implementation

The scikit-learn package contains the **MiniBatchKMeans** algorithm, which allows online learning. This class implements a **partial_fit** function, which takes a set of samples and updates the model. In contrast, calling **fit()** will remove any previous training and refit the model only on the new data.

MiniBatchKMeans follows the same clustering format as other algorithms in scikit-learn, so creating and using it is much the same as other algorithms.

The algorithm works by taking a streaming average of all points that it has seen. To compute this, we only need to keep track of two values, which are the current sum of all seen points, and the number of points seen. We can then use this information, combined with a new set of points, to compute the new averages in the updating step.

Therefore, we can create a matrix X by extracting features from our dataset using `TfIDFVectorizer`, and then sample from this to incrementally update our model. The code is as follows:

```
vec = TfidfVectorizer(max_df=0.4)
X = vec.fit_transform(documents)
```

We then import MiniBatchKMeans and create an instance of it:

```
from sklearn.cluster import MiniBatchKMeans
mbkm = MiniBatchKMeans(random_state=14, n_clusters=3)
```

Next, we will randomly sample from our **X** matrix to simulate data coming in from an external source. Each time we get some data in, we update the model:

```
batch_size = 10
for iteration in range(int(X.shape[0] / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    mbkm.partial_fit(X[start:end])
```

We can then get the labels for the original dataset by asking the instance to predict:

```
labels = mbkm.predict(X)
```

At this stage, though, we can't do this in a pipeline as `TfidfVectorizer` is not an online algorithm. To get over this, we use a `HashingVectorizer`. The `HashingVectorizer` class is a clever use of hashing algorithms to drastically reduce the memory of computing the bag-of-words model. Instead of recording the feature names, such as words found in documents, we record only hashes of those names. This allows us to know our features before we even look at the dataset, as it is the set of all possible hashes. This is a very large number, usually of the order of 2^{18}. Using sparse matrices, we can quite easily store and compute even a matrix of this size, as a very large proportion of the matrix will have the value 0.

Currently, the `Pipeline` class doesn't allow for its use in online learning. There are some nuances in different applications that mean there isn't an obvious one-size-fits-all approach that could be implemented. Instead, we can create our own subclass of **Pipeline**, which allows us to use it for online learning. We first derive our class from **Pipeline**, as we only need to implement a single function:

```
class PartialFitPipeline(Pipeline):
    def partial_fit(self, X, y=None):
        Xt = X
        for name, transform in self.steps[:-1]:
            Xt = transform.transform(Xt)
        return self.steps[-1][1].partial_fit(Xt, y=y)
```

The only function we need to implement is the `partial_fit` function, which is performed by first doing all transformation steps, and then calling partial fit on the final step (which should be the classifier or clustering algorithm). All other functions are the same as in the normal **Pipeline**, class, so we refer (through class inheritance) to those.

We can now create a pipeline to use our `MiniBatchKMeans` in online learning, alongside our `HashingVectorizer`. Other than using our new classes `PartialFitPipeline` and `HashingVectorizer`, this is the same process as used in the rest of this chapter, except we only fit on a few documents at a time. The code is as follows:

```
from sklearn.feature_extraction.text import HashingVectorizer

pipeline = PartialFitPipeline([('feature_extraction', HashingVectorizer()),
                               ('clusterer',
MiniBatchKMeans(random_state=14, n_clusters=3)) ])
batch_size = 10
for iteration in range(int(len(documents) / batch_size)):
    start = batch_size * iteration end = batch_size * (iteration + 1)
    pipeline.partial_fit(documents[start:end])
labels = pipeline.predict(documents)
```

There are some downsides to this approach. For one, we can't easily find out which words are most important for each cluster. We can get around this by fitting another `CountVectorizer` and taking the hash of each word. We then look up values by hash rather than word. This is a bit cumbersome and defeats the memory gains from using HashingVectorizer. Further, we can't use the `max_df` parameter that we used earlier, as it requires us to know what the features mean and to count them over time.

We also can't use tf-idf weighting when performing training online. It would be possible to approximate this and apply such weighting, but again this is a cumbersome approach. `HashingVectorizer` is still a very useful algorithm and a great use of hashing algorithms.

Summary

In this chapter, we looked at clustering, which is an unsupervised learning approach. We use unsupervised learning to explore data, rather than for classification and prediction purposes. In the experiment here, we didn't have topics for the news items we found on reddit, so we were unable to perform classification. We used k-means clustering to group together these news stories to find common topics and trends in the data.

In pulling data from reddit, we had to extract data from arbitrary websites. This was performed by looking for large text segments, rather than a full-blown machine learning approach. There are some interesting approaches to machine learning for this task that may improve upon these results. In the Appendix of this book, I've listed, for each chapter, avenues for going beyond the scope of the chapter and improving upon the results. This includes references to other sources of information and more difficult applications of the approaches in each chapter.

We also looked at a straightforward ensemble algorithm, EAC. An ensemble is often a good way to deal with variance in the results, especially if you don't know how to choose good parameters (which is always difficult with clustering).

Finally, we introduced online learning. This is a gateway to larger learning exercises, including big data, which will be discussed in the final two chapters of this book. These final experiments are quite large and require management of data as well as learning a model from them.

As an extension on the work in this chapter, try implementing EAC to be an online learning algorithm. This is not a trivial task and will involve some thought on what should happen when the algorithm is updated. Another extension is to collect more data from more data sources (such as other subreddits or directly from news websites or blogs) and look for general trends.

In the next chapter, we'll step away from unsupervised learning and go back to classification. We will look at deep learning, which is a classification method built on complex neural networks.

11

Object Detection in Images using Deep Neural Networks

We used basic neural networks in `Chapter 8`, *Beating CAPTCHAs with Neural Networks* with Neural Networks. Research in neural networks is creating some of the most advanced and accurate classification algorithms in many areas. The differences between the concepts introduced in this chapter, versus those introduced in `Chapter 8`, *Beating CAPTCHAs with Neural Networks* is around *complexity*. In this chapter, we look at deep neural networks, those with many hidden layers, and also at more complex layer types for dealing with specific types of information, such as images.

These advances have come on the back of improvements in computational power, allowing us to train larger and more complex networks. However, the advances are much more than simply throwing more computational power at the problem. New algorithms and layer types have drastically improved performance, outside computational power. The cost is that these new classifiers need more data to learn from than other data mining classifiers.

In this chapter, we will look at determining what object is represented in an image. The pixel values will be used as input, and the neural network will then automatically find useful combinations of pixels to form higher-level features. These will then be used for the actual classification.

Overall, in this chapter, we will examine the following:

- Classifying objects in images
- Different types of deep neural networks
- The TensorFlow and Keras libraries to build and train neural networks
- Using a GPU to improve the speed of the algorithms
- Using cloud-based services for added horse-power for data mining

Object classification

Computer vision is becoming an important part of future technology. For example, we will have access to self-driving cars in the very near future - car manufacturers are scheduled to be releasing self-driving models in 2017 and are already partially self-driving. In order to achieve this, the car's computer needs to be able to see around it; identify obstacles, other traffic, and weather conditions; and then use that to plan a safe journey.

While we can easily detect whether there is an obstacle, for example using radar, it is also important we know what that object is. If it is an animal on the road, we can stop and let it move out of the way; if it is a building, this strategy won't work very well!

Use cases

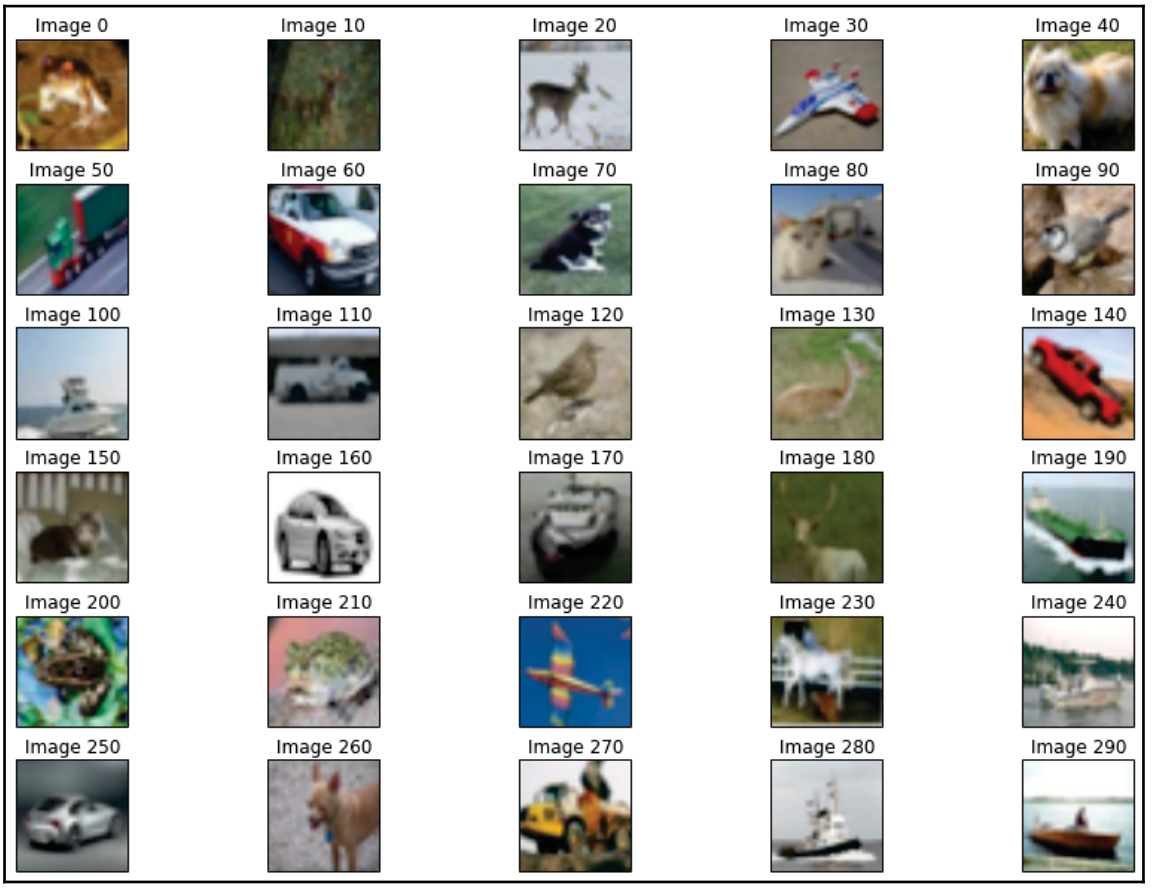
Computer vision is used in many scenarios. Following are some examples where they applications is very important.

- Online map websites, such as Google Maps, use computer vision for a number of reasons. One reason is to automatically blur any faces that they find, in order to give some privacy to the people being photographed as part of their Street View feature.
- Face detection is also used in many industries. Modern cameras automatically detect faces, as a means to improve the quality of photos taken (the user most often wants to focus on a visible face). Face detection can also be used for identification. For example, Facebook automatically recognises people in photos, allowing for easy tagging of friends.
- As we stated before, autonomous vehicles are highly dependent on computer vision to recognise their path and avoid obstacles. Computer vision is one of the key problems that is being addressed not only in research into autonomous vehicles, not just for consumer use, but also in mining and other industries.
- Other industries are using computer vision too, including warehouses examining goods automatically for defects.

- The space industry is also using computer vision, helping to automate the collection of data. This is critical for effective use of spacecraft, as sending a signal from Earth to a rover on Mars can take a long time and is not possible at certain times (for instance, when the two planets are not facing each other). As we start dealing with space-based vehicles more frequently, and from a greater distance, increasing the autonomy of these spacecraft is absolutely necessary and computer vision is a key part of this.The following picture shows the Mars rover designed and used by NASA; it made significant use of computer vision to identify its surroundings on a strange, inhospitable planet.

Application scenario

In this chapter, we will build a system that will take an image as an input and give a prediction on what the object in it is. We will take on the role of a vision system for a car, looking around at any obstacles in the way or on the side of the road. Images are of the following form:



This dataset comes from a popular dataset called CIFAR-10. It contains 60,000 images that are 32 pixels wide and 32 pixels high, with each pixel having a red-green-blue (RGB) value. The dataset is already split into training and testing, although we will not use the testing dataset until after we complete our training.

The CIFAR-10 dataset is available for download at `http://www.cs.toronto.edu/~kriz/cifar.html`
Download the python version, which has already been converted to NumPy arrays.

Opening a new Jupyter Notebook, we can see what the data looks like. First, we set up the data filenames. We will only worry about the first batch to start with, and scale up to the full dataset size towards the end;

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "cifar-10-
batches-py")
batch1_filename = os.path.join(data_folder, "data_batch_1")
```

Next, we create a function that can read the data stored in the batches. The batches have been saved using pickle, which is a python library to save objects. Usually, we can just call `pickle.load(file)` on the file to get the object. However, there is a small issue with this data: it was saved in Python 2, but we need to open it in Python 3. In order to address this, we set the encoding to `latin` (even though we are opening it in byte mode):

```
import pickle
# Bugfix thanks to:
http://stackoverflow.com/questions/11305790/pickle-incompatability-of-numpy
-arrays-between-python-2-and-3
def unpickle(filename):
    with open(filename, 'rb') as fo:
        return pickle.load(fo, encoding='latin1')
```

Using this function, we can now load the batch dataset:

```
batch1 = unpickle(batch1_filename)
```

This batch is a dictionary containing the actual data in NumPy arrays, the corresponding labels and filenames, and a note to say which batch it is (this is training batch 1 of 5, for instance).

We can extract an image by using its index in the batch's data key:

```
image_index = 100
image = batch1['data'][image_index]
```

The image array is a NumPy array with 3,072 entries, from 0 to 255. Each value is the red, green, or blue intensity at a specific location in the image.

The images are in a different format than what matplotlib usually uses (to display images), so to show the image we first need to reshape the array and rotate the matrix. This doesn't matter so much to train our neural network (we will define our network in a way that fits with the data), but we do need to convert it for **matplotlib's** sake:

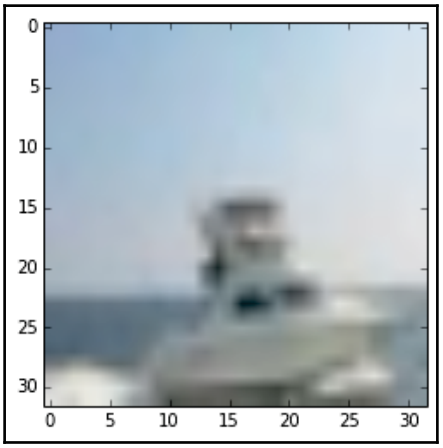
```
image = image.reshape((32,32, 3), order='F')
import numpy as np
image = np.rot90(image, -1)
```

Now we can show the image using matplotlib:

```
%matplotlib inline

from matplotlib import pyplot as plt
plt.imshow(image)
```

The resulting image, a boat, is displayed:



The resolution of this image is quite poor—it is only 32 pixels wide and 32 pixels high. Despite that, most people will look at the image and see a boat. Can we get a computer to do the same?

You can change the image index to show different images, getting a feel for the dataset's properties.

The aim of our project, in this chapter, is to build a classification system that can take an image like this and predict what the object in it is. Before we do that though, we will take a detour to learn about the classifier we are going to use: **Deep neural networks**.

Deep neural networks

The neural networks we used in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*, have some fantastic *theoretical* properties. For example, only a single hidden layer is needed to learn any mapping (although the size of the middle layer may need to be very, very big). Neural networks were a very active area of research in the 1970s and 1980s due to this theoretical perfection. However several issues caused them to fall out of favor, particularly compared to other classification algorithms such as support vector machines. A few of the major ones are listed here:

- One of the main issues was that the computational power needed to run many neural networks was more than other algorithms and more than what many people had access to.
- Another issue was training the networks. While the back propagation algorithm has been known about for some time, it has issues with larger networks, requiring a very large amount of training before the weights settle.

Each of these issues has been addressed in recent times, leading to a resurgence in popularity of neural networks. Computational power is now much more easily available than 30 years ago, and advances in algorithms for training mean that we can now readily use that power.

Intuition

The aspect that differentiates **deep neural networks** from the more basic neural network we saw in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*, is size.

A neural network is considered deep when it has two or more hidden layers. In practice, a deep neural network is often much larger, both in the number of nodes in each layer and also the number of layers. While some of the research of the mid -2000s focused on very large numbers of layers, smarter algorithms are reducing the actual number of layers needed.

The size is one differentiator, but new layer types and neural network structures are assisting in creating deep neural networks for specific areas. We have already seen a feed-forward neural network composed of **dense layers**. This means we have a series of layers, in order, where each neuron from one layer is attached to each neuron from another layer. Other types include:

- **Convolutional Neural Networks** (**CNN**) for image analysis. In this case, a small segment of the image is taken as a single input, and that input is passed onto a pooling layer to combine these outputs. This helps with issues such as rotation and translation of images. We will use these networks in this chapter.
- **Recurrent Neural Networks** (**RNN**) for text and time-series analysis. In this case, the previous state of the neural network is remembered and used to alter the current output. Think of the preceding word in a sentence modifying the output for the current word in the phrase: *United States*. One of the most popular types is an LSTM recurrent network, standing for **Long-Short Term Memory**.
- **Autoencoders**, which learn a mapping from the input, through a hidden layer (usually with fewer nodes), back to the input. This finds a compression of the input data, and this layer can be reused in other neural networks, reducing the amount of labelled training data needed.

There are many, many more types of neural networks. Research into applications and theory of deep neural networks is finding more and more forms of neural networks every month. Some are designed for general purpose learning, some for specific tasks. Further, there are multiple ways to combine layers, tweak parameters, and otherwise alter the learning strategy. For example, **dropout layers** randomly reduce some weights to zero during training, forcing all parts of the neural network to learn good weights.

Despite all these differences, a neural network is usually designed to take very basic features as inputs—in the case of computer vision, it is simple pixel values. As that data is combined and pushed through the network, these basic features combine into more complex features. Sometimes, these features have little meaning to humans, but they represent the aspects of the sample that the computer looks for to make its classification.

Implementing deep neural networks

Implementing these deep neural networks can be quite challenging due to their size. A bad implementation will take significantly longer to run than a good one, and may not even run at all due to memory usage.

A basic implementation of a neural network might start by creating a node class and collecting a set of these into a layer class. Each node is then connected to a node in the next layer using an instance of an *Edge* class. This type of implementation, a class-based one, is good to show how networks operate but too inefficient for larger networks. Neural networks simply have too many moving parts for this strategy to be efficient.

Instead, most neural networks operations can be expressed as mathematical expressions on matrices. The weights of the connections between one network layer and the next can be represented as a matrix of values, where the rows represent nodes in the first layer and the columns represent the nodes in the second layer (the transpose of this matrix is used sometimes too). The value is the weight of the edge between one layer and the next. A network can then be defined as a set of these weight matrices. In addition to the nodes, we add a bias term to each layer, which is basically a node that is always on and connected to each neuron in the next layer.

This insight allows us to use matrix operations to build, train, and use neural networks, as opposed to creating a class-based implementation. These mathematical operations are great, as many great libraries of highly optimised code have been written that we can use to perform these computations as efficiently as we can.

The scikit-learn implementation that we used in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*, does contain some features for building neural networks but lacks several recent advances in the field. For larger and more customised networks, though, we need a library that gives us a bit more power. We will use the **Keras** library instead to create our deep neural network.

In this chapter, we will start by implementing a basic neural network with Keras and then (nearly) replicate our experiment in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*, on predicting which letter is in an image. Finally, we will use a much more complex convolution neural network to perform image classification on the CIFAR dataset, which will also include running this on GPUs rather than CPUs to improve the performance.

Keras is a high-level interface to using a graph-computation library for implementing deep neural networks. Graph-computation libraries outline a series of operations and then later compute the values. These are great for matrix operations because they can be used to represent data flows, distribute those data flows across multiple systems and perform other optimisations. Keras can use either of two graph-computation libraries under the hood. The first is called **Theano**, which is a little older and has a strong following (and was used in the first edition of this book), and the second is **TensorFlow**, released recently by Google and is the library that powers much of their deep learning. Ultimately, you can use either library in this chapter.

An Introduction to TensorFlow

TensorFlow is a graph computation library designed by engineers at Google, and is starting to power many of Google's recent advances in **deep learning** and **artificial intelligence**.

A graph computation library has two steps. They are listed below:

1. Defining the sequence (or more complex graphs) of operations that take the input data, operate on it, and convert to outputs.
2. Compute on the graph obtained from step 1 with a given input.

Many programmers don't use this type of programming day-to-day, but most of them interact with a related system that does. Relational databases, specifically SQL-based ones, use a similar concept called the declarative paradigm. While a programmer might define a `SELECT` query on a database with a `WHERE` clause, the database interprets that and creates an optimised query based on a number of factors, such as whether the `WHERE` clause is applied to a primary key, the format the data is stored in, and other factors. The programmer defines what they want and the system determines how to do it.

You can install TensorFlow using Anaconda: **conda install tensorflow**
For more options, Google has a detailed installation page at `https://www.tensorflow.org/get_started/os_setup`

Using TensorFlow, we can define many types of functions working on scalars, arrays, and matrices, as well as other mathematical expressions. For instance, we can create a graph that computes the values of a given quadratic equation:

```
import tensorflow as tf

# Define the parameters of the equation as constant values
a = tf.constant(5.0)
```

```
b = tf.constant(4.5)
c = tf.constant(3.0)

# Define the variable x, which lets its value be changed
x = tf.Variable(0., name='x')  # Default of 0.0

# Define the output y, which is an operation on a, b, c and x
y = (a * x ** 2) + (b * x) + c
```

This *y* object is a Tensor object. It does not yet have a value as this hasn't been computed. All we have done is create a graph that states:

When we do compute **y**, *first take the square the value of* **x** *and multiply it by* **a**, *add* **b** *times* **x** *to it, and then add* **c** *to the result.*

The graph itself can be viewed through TensorFlow. Here is some code to visualise this graph within a Jupyter Notebook, courtesy of StackOverflow user Yaroslav Bulatov (see this answer: `http://stackoverflow.com/a/38192374/307363`):

```
from IPython.display import clear_output, Image, display, HTML

def strip_consts(graph_def, max_const_size=32):
    """Strip large constant values from graph_def."""
    strip_def = tf.GraphDef()
    for n0 in graph_def.node:
        n = strip_def.node.add()
        n.MergeFrom(n0)
        if n.op == 'Const':
            tensor = n.attr['value'].tensor
            size = len(tensor.tensor_content)
            if size > max_const_size:
                tensor.tensor_content = "<stripped %d bytes>"%size
    return strip_def

def show_graph(graph_def, max_const_size=32):
    """Visualize TensorFlow graph."""
    if hasattr(graph_def, 'as_graph_def'):
        graph_def = graph_def.as_graph_def()
    strip_def = strip_consts(graph_def, max_const_size=max_const_size)
    code = """
        <script>
          function load() {{
            document.getElementById("{id}").pbtxt = {data};
          }}
        </script>
        <link rel="import"
href="https://tensorboard.appspot.com/tf-graph-basic.build.html"
onload=load()>
```

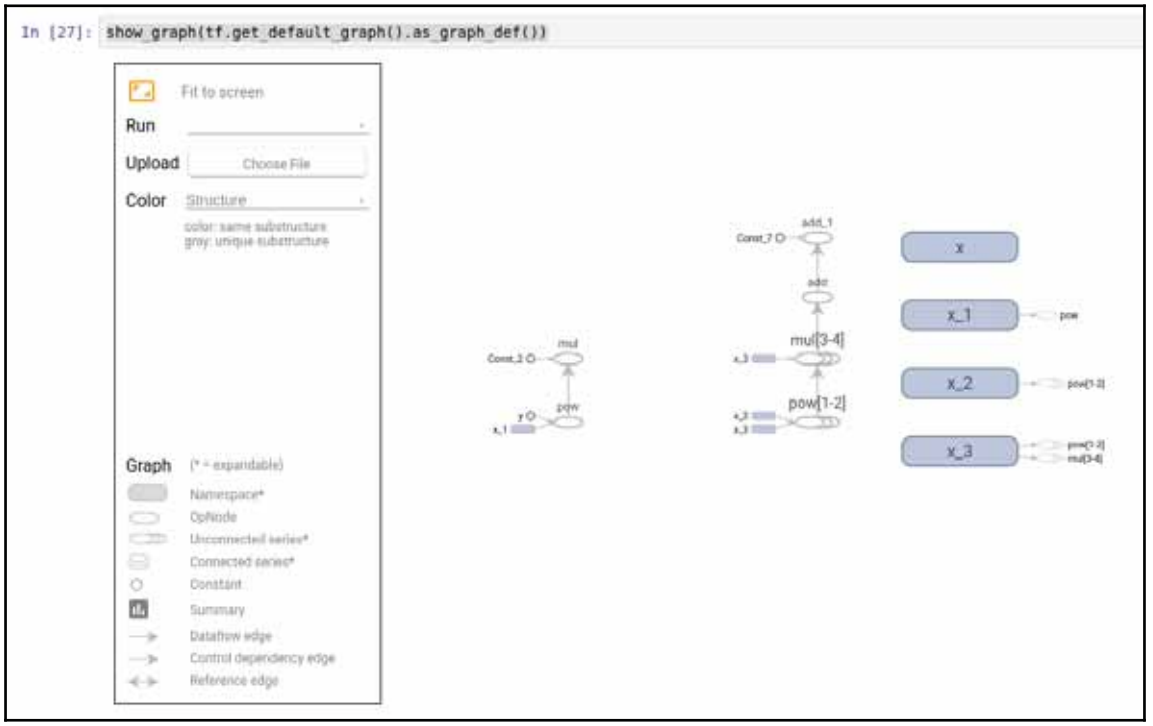
```
            <div style="height:600px">
              <tf-graph-basic id="{id}"></tf-graph-basic>
            </div>
        """.format(data=repr(str(strip_def)), id='graph'+str(np.random.rand()))

        iframe = """
            <iframe seamless style="width:1200px;height:620px;border:0"
    srcdoc="{}"></iframe>
        """.format(code.replace('"', '&quot;'))
        display(HTML(iframe))
```

We can then perform the actual visualisation using this code in a new cell:

```
show_graph(tf.get_default_graph().as_graph_def())
```

The results show how these operations are linked in a directed graph. The visualisation platform is called **TensorBoard**, which comes with TensorFlow:



When we want to compute a value for **y**, we need to pass a value for **x** through the other nodes in the graph, these are called **OpNodes** in the above graph, short for *Operation Node*.

To this point, we have defined the graph itself. The next step is to compute the values. We can do this a number of ways, especially considering **x** is a **Variable**. To compute **y**, using the current value of **x**, we create a TensorFlow Session object and then ask it to run **y**:

```
model = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(model)
    result = session.run(y)
print(result)
```

The first line initialises the variables. TensorFlow lets you specify scopes of operations and namespaces. At this point, we are just using the global namespace, and this function is a handy shortcut to initialise that scope properly, which can be thought of as a step needed for TensorFlow to compile the graph.

The second creates a new session that will run the model itself. The result from `tf.global_variables_initializer()` is itself an operation on the graph, and must be executed to happen. The next line actually runs the variable **y**, which computes the necessary **OpNodes** needed to compute the value of **y**. In our case, that is all of the nodes but it is possible that larger graphs might not need all nodes computed - TensorFlow will do just enough work to get the answer and no more.

If you get an error that `global_variables_initializer` is not defined, replace it with `initialize_all_variables` - the interface was recently changed.

Printing the result gives us our value of 3.

We can also do other operations, such as change the value of **x**. For instance, we can create an assign operation, which assigns a new value to an existing Variable. In this example, we change the value of **x** to 10 and then compute y, which results in 548.

```
model = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(model)
    session.run(x.assign(10))
    result = session.run(y)
print(result)
```

While this simple example may not seem much more powerful than what we can already do with Python, TensorFlow (and Theano) have large amounts of distribution options for computing larger networks over many computers and optimisations for doing it efficiently. Both libraries also contain extra tools for saving and loading networks, including values, which lets us save models created in these libraries.

Using Keras

TensorFlow is not a library to directly build neural networks. In a similar way, NumPy is not a library to perform data mining; it just does the heavy lifting and is generally used from another library. TensorFlow contains a built-in library, referred to as TensorFlow Learn to build networks and perform data mining. Other libraries, such as Keras, are also built with this in mind and use TensorFlow in the backend.

Keras implements a number of modern types of neural network layers and the building blocks for building them. In this chapter, we will use convolution layers which are designed to mimic the way in which human vision works. They use small collections of connected neurons that analyse only a segment of the input values - in this case, an image. This allows the network to deal with standard alterations such as dealing with translations of images. In the case of vision-based experiments, an example of an alteration dealt with by convolution layers is translating the image.

In contrast, a traditional neural network is often heavily connected—all neurons from one layer connect to all neurons in the next layer. This is referred to as a dense layer.

The standard model for neural networks in Keras is a **Sequential** model, which is created by passing a list of layers. The input (**X_train**) is given to the first layer, and its output given to the next layer and so on, in a standard feed-forward configuration.

Building a neural network in Keras is significantly easier than building it using just TensorFlow. Unless you are doing highly customised modifications to the neural network structure, I strongly recommend using Keras.

To show the basics of using Keras for neural networks, we will implement a basic network to lean on the Iris dataset, which we saw in `Chapter 1`, *Getting Started with Data Mining*. The Iris dataset is great for testing new algorithms, even complex ones such as deep neural networks.

First, open a new Jupyter Notebook. We will come back to the Notebook with the CIFAR data, later in the chapter.

Next, we load the dataset:

```
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data.astype(np.float32)
y_true = iris.target.astype(np.int32)
```

When dealing with libraries like TensorFlow, it is best to be quite explicit about data types. While Python will happily convert from one numerical data type to another implicitly, libraries like TensorFlow are wrappers around lower-level code (in this case, C++). These libraries are not able to always convert between numerical data types.

Our output is currently a single array of categorical values (0, 1 or 2 depending on the class). Neural networks *can* be developed to output data in this format, but the *normal convention* is for the neural network to have *n* outputs, where *n* in the number of classes. Due to this, we use one-hot encoding to convert our categorical **y** into a one-hot encoded `y_onehot`:

```
from sklearn.preprocessing import OneHotEncoder

y_onehot = OneHotEncoder().fit_transform(y_true.reshape(-1, 1))
y_onehot = y_onehot.astype(np.int64).todense()
```

We then split into training and testing datasets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot,
random_state=14)
```

Next, we build our network by creating the different layers. Our dataset contains four input variables and three output classes. This gives us the size of the first and last layer, but not the layers in between. Playing around with this figure will give different results, and it is worth trailing different values to see what happens. We will create a small network to start with, with the following dimensions:

```
input_layer_size, hidden_layer_size, output_layer_size = 4, 6, 3
```

Next, we create our hidden layer and our output layer (the input layer is implicit). For this example we will use Dense layers:

```
from keras.layers import Dense
hidden_layer = Dense(output_dim=hidden_layer_size,
input_dim=input_layer_size, activation='relu')
output_layer = Dense(output_layer_size, activation='sigmoid')
```

I encourage you to play with the activation value, and see how that affects the results. The values here are great defaults if you have no further information about your problem. That is, use `relu` for hidden layers, and `sigmoid` for the output layer.

We then combine the layers into a Sequential model:

```
from keras.models import Sequential
model = Sequential(layers=[hidden_layer, output_layer])
```

One necessary step from here is to compile the network, which creates the graph. In the compile step, we were given information on how the network will be trained and evaluated. The values here define what exactly it is that the neural network is trying to train to reduce, in the case below, it is the mean squared error between the output neurons and their expected values. The choice of optimizer largely affects how efficiently it can do this, often with a trade-off between speed and memory usage.

```
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['accuracy'])
```

We then train our model using the `fit` function. Keras models return a history object from `fit()`, that allows us see the data at a fine-grained level.

```
history = model.fit(X_train, y_train)
```

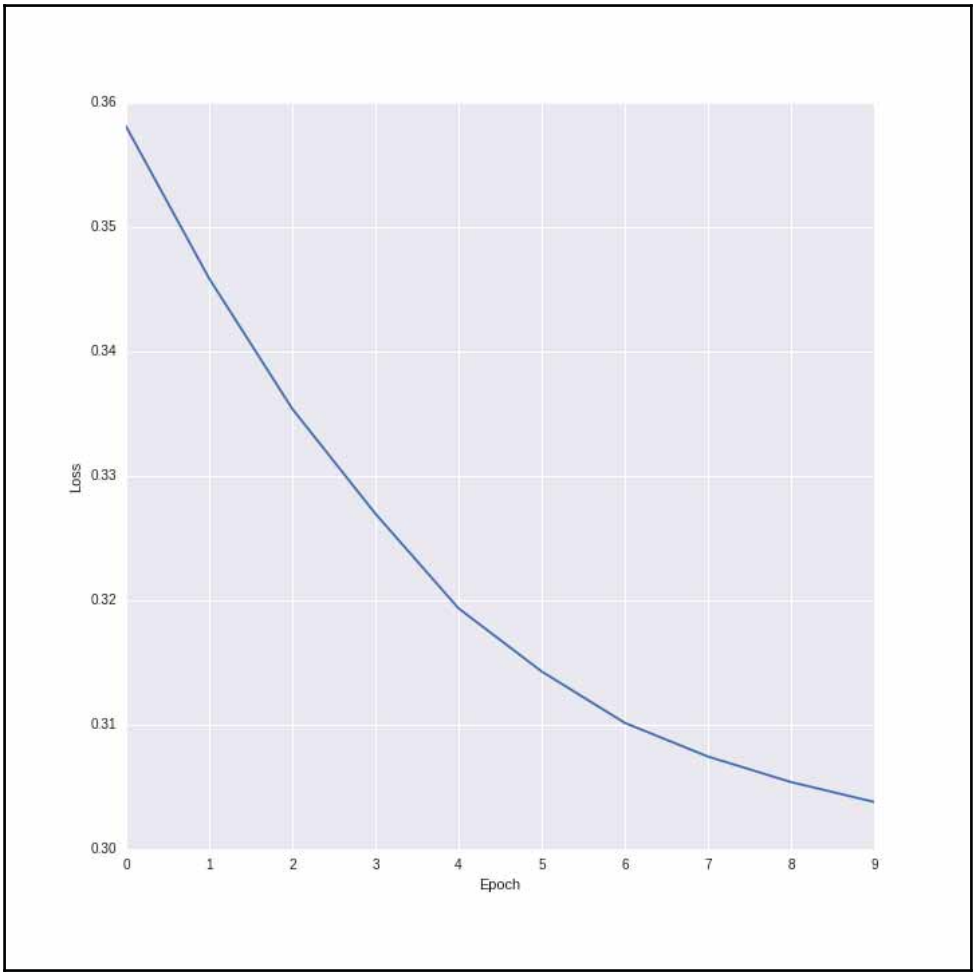
You will get quite a lot of output. The neural network will train 10 epochs, which are training cycles of taking the training data, running it through the neural network, updating the weights and evaluating the results. If you investigate the history object (try `print(history.history)`) you will see the loss function's score after each of these epochs (lower is better). Also included is the accuracy, where higher is better. You will probably also notice that it hasn't really improved that much.

We can plot out the history object using `matplotlib`:

```
import seaborn as sns
from matplotlib import pyplot as plt

plt.plot(history.epoch, history.history['loss'])
```

```
plt.xlabel("Epoch")
plt.ylabel("Loss")
```



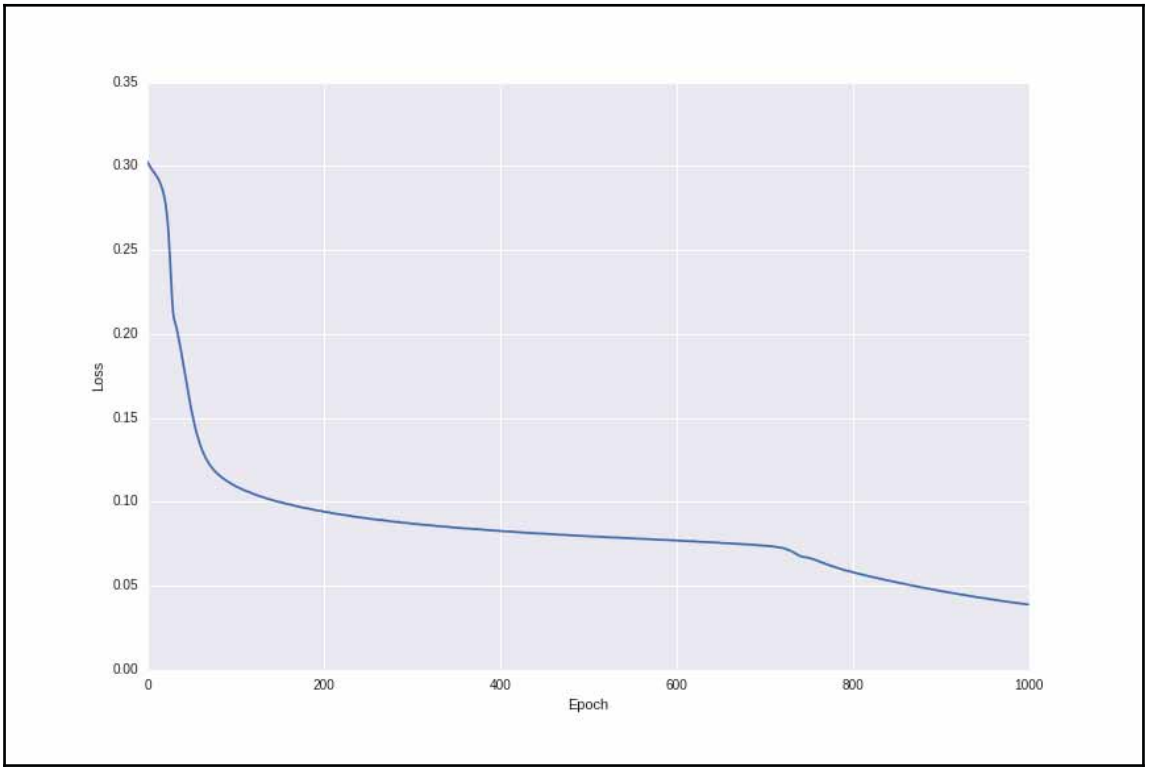
While the training loss is decreasing, it is not decreasing much. This is one issue with neural networks - they train slowly. By default, the fit function will only perform 10 epochs, which is nowhere near enough for nearly any application. To see this, use the neural network to predict the test set and run a classification report:

```
from sklearn.metrics import classification_report
y_pred = model.predict_classes(X_test)
print(classification_report(y_true=y_test.argmax(axis=1), y_pred=y_pred))
```

The results are quite poor, with an overall f1-score of 0.07, and the classifier only predicting class 2 for all instances. At first, it might seem that neural networks are not that great but let's have a look at what happens when we train for 1000 epochs:

```
history = model.fit(X_train, y_train, nb_epoch=1000, verbose=False)
```

Visualizing the loss per epoch again, a very useful visualization when running iterative algorithms like neural networks, using the above code shows a very different story:



Finally, we perform a classification report again to see the results:

```
y_pred = model.predict_classes(X_test)
print(classification_report(y_true=y_test.argmax(axis=1), y_pred=y_pred))
```

Perfect.

Convolutional Neural Networks

To get started with image analysis with Keras, we are going to reimplement the example we used in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*, to predict which letter was represented in an image. We will recreate the dense neural network we used in `Chapter 8`, *Beating CAPTCHAs with Neural Networks*. To start with, we need to enter our dataset building code again in our notebook. For a description of what this code does, refer to `Chapter 8`, *Beating CAPTCHAs with Neural Networks* (*remember to update the file location of the Coval font*):

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf

def create_captcha(text, shear=0, size=(100, 30), scale=1):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(r"bretan/Coval-Black.otf", 22)
    draw.text((0, 0), text, fill=1, font=font)
    image = np.array(im)
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    image = image / image.max()
    shape = image.shape
    # Apply scale
    shapex, shapey = (shape[0] * scale, shape[1] * scale)
    image = tf.resize(image, (shapex, shapey))
    return image

from skimage.measure import label, regionprops
from skimage.filters import threshold_otsu
from skimage.morphology import closing, square

def segment_image(image):
    # label will find subimages of connected non-black pixels
    labeled_image = label(image>0.2, connectivity=1, background=0)
    subimages = []
    # regionprops splits up the subimages
    for region in regionprops(labeled_image):
        # Extract the subimage
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x,start_y:end_y])
    if len(subimages) == 0:
        # No subimages found, so return the entire image
        return [image,]
    return subimages
```

```
from sklearn.utils import check_random_state
random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
assert len(letters) == 26
shear_values = np.arange(0, 0.8, 0.05)
scale_values = np.arange(0.9, 1.1, 0.1)

def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
    scale = random_state.choice(scale_values)
    return create_captcha(letter, shear=shear, size=(30, 30), scale=scale),
letters.index(letter)

dataset, targets = zip(*(generate_sample(random_state) for i in
range(1000)))
dataset = np.array([tf.resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
dataset = np.array(dataset, dtype='float')
targets = np.array(targets)

from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
y = y.todense()

X = dataset.reshape((dataset.shape[0], dataset.shape[1] *
dataset.shape[2]))

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.9)
```

After rerunning all of this code, you'll have a dataset similar to `Chapter 8`, *Beating CAPTCHAs with Neural Networks* experiment. Next, instead of using scikit-learn to do our neural network, we will use Keras.

First, we create our two **Dense** layers and combine them in a **Sequential** model. I've chosen to put 100 neurons in the hidden layer.

```
from keras.layers import Dense
from keras.models import Sequential
hidden_layer = Dense(100, input_dim=X_train.shape[1])
output_layer = Dense(y_train.shape[1])
# Create the model
model = Sequential(layers=[hidden_layer, output_layer])
model.compile(loss='mean_squared_error', optimizer='adam',
metrics=['accuracy'])
```

Then, we fit the model. As before, you will want to have quite a larger number of epochs. I've used 1000 again, if you want better results, you can increase this number.

```
model.fit(X_train, y_train, nb_epoch=1000, verbose=False)
y_pred = model.predict(X_test)
```

You can also collect the resulting history object, like we did with the Iris example, to investigate the training further.

```
from sklearn.metrics import classification_report
print(classification_report(y_pred=y_pred.argmax(axis=1),
y_true=y_test.argmax(axis=1)))
```

Again, perfect.

At least, it was on my machine but your results may differ slightly.

GPU optimization

Neural networks can grow quite large in size. This has some implications for memory use; however, efficient structures such as sparse matrices mean that we don't generally run into problems fitting a neural network in memory.

The main issue when neural networks grow large is that they take a very long time to compute. In addition, some datasets and neural networks will need to run many epochs of training to get a good fit for the dataset.

The neural network we will train in this chapter takes more than 8 minutes per epoch on my reasonably powerful computer, and we expect to run dozens, potentially hundreds, of epochs. Some larger networks can take hours to train a single epoch. To get the best performance, you may be considering thousands of training cycles.

The scale of neural networks leads to long training times.

One positive is that neural networks are, at their core, full of floating point operations. There are also a large number of operations that can be performed in parallel, as neural network training is composed of mainly matrix operations. These factors mean that computing on GPUs is an attractive option to speed up this training.

When to use GPUs for computation

GPUs were originally designed to render graphics for display. These graphics are represented using matrices and mathematical equations on those matrices, which are then converted into the pixels that we see on our screen. This process involves lots of computation in parallel. While modern CPUs may have a number of cores (your computer may have 2, 4, or even 16—or more!), GPUs have thousands of small cores designed specifically for graphics.

A CPU is better for sequential tasks, as the cores tend to be individually faster and tasks such as accessing the computer's memory are more efficient. It is also, honestly, easier to just let the CPU do the heavy lifting. Almost every machine learning library defaults to using the CPU, and there is extra work involved before you can use the GPU for computing. The benefits can be quite significant.

GPUs are therefore better suited for tasks in which there are lots of small operations on numbers that can be performed at the same time. Many machine learning tasks are like this, lending themselves to efficiency improvements through the use of a GPU.

Getting your code to run on a GPU can be a frustrating experience. It depends greatly on what type of GPU you have, how it is configured, your operating system, and whether you are prepared to make some low-level changes to your computer.

Luckily, Keras will automatically use a GPU for operations, if the operation suits and a GPU can be found (and if you use TensorFlow as the backend). However, you still need to setup your computer such that the GPU can be found by Keras and TensorFlow.

There are three main avenues to take:

- The first is to look at your computer, search for tools and drivers for your GPU and operating system, explore some of the many tutorials out there, and find one that fits your scenario. Whether this works depends on what your system is like. That said, this scenario is much easier than it was a few years ago, with better tools and drivers available to perform GPU-enabled computation.
- The second avenue is to choose a system, find good documentation on setting it up and buy a system to match. This will work better, but can be fairly expensive—in most modern computers, the GPU is one of the most expensive parts. This is especially true if you want to get great performance out of the system—you'll need a really good GPU, which can be very expensive. If you are a business (or have larger amounts of money to spend), you can buy high-end GPUs specifically for deep learning and talk more directly to vendors to ensure you get the right hardware.

- The third avenue is to use a virtual machine, which is already configured for such a purpose. For example, Altoros Systems has created such a system that runs on Amazon's Web Services. The system will cost you money to run, but the price is much less than that of a new computer. Depending on your location, the exact system you get and how much you use it, you are probably looking at less than $1 an hour, and often much, much less. If you use spot instances in Amazon's Web Services, you can run them for just a few cents per hour (although, you will need to develop your code to run on spot instances separately).

If you aren't able to afford the running costs of a virtual machine, I recommend that you look into the first avenue, with your current system. You may also be able to pick up a good second-hand GPU from family or a friend who constantly updates their computer (gamer friends are great for this!).

Running our code on a GPU

We are going to take the third avenue in this chapter and create a virtual machine based on Altoros Systems' base system. This will run on an Amazon's EC2 service. There are many other Web services to use, and the procedure will be slightly different for each. In this section, I'll outline the procedure for Amazon.

If you want to use your own computer and have it configured to run GPU-enabled computation, feel free to skip this section.

You can get more information on how this was set up, see `https://aws.amazon.com/marketplace/pp/B01H1VWUOY?qid=1485755720051&sr=0-1&ref_=srh_res_product_title`

1. To start with, go to the AWS console at: `https://console.aws.amazon.com/console/home?region=us-east-1`
2. Log in with your Amazon account. If you don't have one, you will be prompted to create one, which you will need to do in order to continue.
3. Next, go to the EC2 service console at: `https://console.aws.amazon.com/ec2/v2/home?region=us-east-1`.
4. Click on **Launch Instance** and choose N. California as your location in the drop-down menu at the top-right.

5. Click on **Community AMIs** and search for **Ubuntu x64 AMI with TensorFlow (GPU)**, which is the machine created by Altoros Systems. Then, click on **Select**. On the next screen, choose **g2.2xlarge** as the machine type and click on **Review and Launch**. On the next screen, click on **Launch**.
6. At this point, you will be charged, so please remember to shut down your machines when you are done with them. You can go to the EC2 service, select the machine, and stop it. You won't be charged for machines that are not running.
7. You'll be prompted with some information on how to connect to your instance. If you haven't used AWS before, you will probably need to create a new key pair to securely connect to your instance. In this case, give your key pair a name, download the pemfile, and store it in a safe place—if lost, you will not be able to connect to your instance again!
8. Click on **Connect** for information on using the **pem** file to connect to your instance. The most likely scenario is that you will use ssh with the following command:

```
ssh –i <certificante_name>.pem ubuntu@<server_ip_address>
```

Setting up the environment

Next, we need to get our code onto the machine. There are many ways to get this file onto your computer, but one of the easiest is to just copy-and-paste the contents.

To start with, open the Jupyter Notebook we used before (on your computer, not on the Amazon Virtual Machine). On the Notebook itself is a menu. Click on **File** and then **Download as**. Select **Python** and save it to your computer. This procedure downloads the code in the Jupyter Notebook as a python script that you can run from the command line.

Open this file (on some systems, you may need to right-click and open with a text editor). Select all of the contents and copy them to your clipboard.

On the Amazon Virtual Machine, move to the home directory and open **nano** with a new filename:

```
$ cd~/

$ nano chapter11script.py
```

The nano program will open, which is a command-line text editor.

With this program open, paste the contents of your clipboard into this file. On some systems, you may need to use a file option of the ssh program, rather than pressing Ctrl+ V to paste.

In nano, press Ctrl+ O to save the file on the disk and then Ctrl+ X to exit the program.

You'll also need the font file. The easiest way to do this is to download it again from the original location. To do this, enter the following:

```
$ wget
http://openfontlibrary.org/assets/downloads/bretan/680bc56bbeeca95353ede363
a3744fdf/bretan.zip

$ sudo apt-get install unzip

$ unzip -p bretan.zip
```

While still in the virtual machine, you can run the program with the following command:

```
$ python chapter11script.py
```

The program will run through as it would in the Jupyter Notebook and the results will print to the command line.

The results should be the same as before, but the actual training and testing of the neural network will be much faster. Note that it won't be that much faster in the other aspects of the program—we didn't write the CAPTCHA dataset creation to use a GPU, so we will not obtain a speedup there.

You may wish to shut down the Amazon virtual machine to save some money; we will be using it at the end of this chapter to run our main experiment, but will be developing the code on your main computer first.

Application

Back on your main computer now, open the first Jupyter Notebook we created in this chapter—the one that we loaded the CIFAR dataset with. In this major experiment, we will take the CIFAR dataset, create a deep convolution neural network, and then run it on our GPU-based virtual machine.

Getting the data

To start with, we will take our CIFAR images and create a dataset with them. Unlike previously, we are going to preserve the pixel structure—that is, in rows and columns. First, load all the batches into a list:

```
import os
import numpy as np

data_folder = os.path.join(os.path.expanduser("~"), "Data", "cifar-10-
batches-py")

batches = []
for i in range(1, 6):
    batch_filename = os.path.join(data_folder, "data_batch_{}".format(i))
    batches.append(unpickle(batch_filename))
    break
```

The last line, the break, is to test the code—this will drastically reduce the number of training examples, allowing you to quickly see if your code is working. I'll prompt you later to remove this line after you have tested that the code works.

Next, create a dataset by stacking these batches on top of each other. We use NumPy's **vstack**, which can be visualised as adding rows to the end of the array:

```
X = np.vstack([batch['data'] for batch in batches])
```

We then normalise the dataset to the range 0 to 1 and then force the type to be a 32-bit float (this is the only datatype the GPU-enabled virtual machine can run with):

```
X = np.array(X) / X.max()
X = X.astype(np.float32)
```

We then do the same with the classes, except we perform a **hstack**, which is similar to adding columns to the end of the array. We could then use the **OneHotEncoder** to turn this into a one-hot array. I'll show an alternate method here using a utility function present in Keras, but the result is the same either way:

```
from keras.utils import np_utils
y = np.hstack(batch['labels'] for batch in batches).flatten()
nb_classes = len(np.unique(y))
y = np_utils.to_categorical(y, nb_classes)
```

Next, we split the dataset into training and testing sets:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Next, we reshape the arrays to preserve the original data structure. The original data was 32-by-32-pixel images, with 3 values per pixel (for the red, green, and blue values). While standard feed-forward neural networks only take a single array of input data (see the CAPTCHA example), Convolutional Neural Networks are built for images and accept 3-dimensional image data (2-D image, and another dimension containing colour depth).

```
X_train = X_train.reshape(-1, 3, 32, 32)
X_test = X_test.reshape(-1, 3, 32, 32)
n_samples, d, h, w = X_train.shape  # Obtain dataset dimensions
# Convert to floats and ensure data is normalised.
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

We now have a familiar training and testing dataset, along with the target classes for each. We can now build the classifier.

Creating the neural network

We will now build the convolutional neural network. I have performed some tinkering and found a layout that works well, but feel free to experiment with more layers (or fewer), layers of different types and different sizes. Smaller networks train faster, but larger networks can achieve better results.

First, we create the layers of our neural network:

```
from keras.layers import Dense, Flatten, Convolution2D, MaxPooling2D
conv1 = Convolution2D(32, 3, 3, input_shape=(d, h, w), activation='relu')
pool1 = MaxPooling2D()
conv2 = Convolution2D(64, 2, 2, activation='relu')
pool2 = MaxPooling2D()
conv3 = Convolution2D(128, 2, 2, activation='relu')
pool3 = MaxPooling2D()
flatten = Flatten()
hidden4 = Dense(500, activation='relu')
hidden5 = Dense(500, activation='relu')
output = Dense(nb_classes, activation='softmax')
layers = [conv1, pool1,
          conv2, pool2,
          conv3, pool3,
          flatten, hidden4, hidden5,
          output]
```

We use dense layers for the last three layers as per a normal feed-forward neural network, but before that, we use convolution layers combined with pooling layers. We have three sets of these.

For each pair of **Convolution2D** and **MaxPooling2D** layers, the following happens:

1. The **Convolution2D** network fetches patches of the input data. These are passed through a filter, a matrix transformation akin to the kernel operator Support Vector Machines use. A filter is a smaller matrix, of size **k** by **n** (specified as 3x3 in the Convolution2D initialiser above) that is applied to each **k** by **n** pattern found in the image. The result is a convolved feature.
2. The **MaxPooling2D** layer takes the results from the **Convolution2D** layer and finds the maximum value for each convolved feature.

While this does discard lots of information, this actually helps for image detection. If the object of an image is a few pixels to the right, a standard neural network will consider it a completely new image. In contrast, the convolution layer will find it and report almost the same output (depending, of course, on a wide variety of other factors).

After passing through these pairs layers, the features that go into the dense part of the network are meta-features that represent abstract concepts of the image, rather than specific qualities. Often these can be visualised, resulting in features like *a little bit of a line pointing up*.

Next, we put these layers together to build our neural network and train it. This training will take substantially longer than previous training. I recommend starting with 10 epochs, make sure the code works all the way through, and then rerun with 100 epochs. Also, once you have confirmed that the code works and you get predictions out, go back and remove the `break` line we put in when creating the dataset (it is in the batches loop). This will allow the code to train on all of the samples, not just the first batch.

```
model = Sequential(layers=layers)
model.compile(loss='mean_squared_error', optimizer='adam',
metrics=['accuracy'])
import tensorflow as tf
history = model.fit(X_train, y_train, nb_epoch=25, verbose=True,
validation_data=(X_test, y_test),batch_size=1000))
```

Finally, we can predict with the network and evaluate.

```
y_pred = model.predict(X_test)
from sklearn.metrics import classification_report
print(classification_report(y_pred=y_pred.argmax(axis=1),
 y_true=y_test.argmax(axis=1)))
```

After running for 100 epochs, it is still not quite perfect in this case, but still an excellent result. If you have the time (say, overnight), try running the code for 1000 epochs. There is an increase in accuracy but a diminishing return on time invested. A (not so) good rule of thumb is that to halve the error, you need to double the training time.

Putting it all together

Now that we have our network code working, we can train it with our training dataset on the remote machine. If you used your local machine to run the neural network, you can skip this section.

We need to upload the script to our virtual machine. As with before, click on **File| Download** as, Python, and save the script somewhere on your computer. Launch and connect to the virtual machine and upload the script as you did earlier (I called my script `chapter11cifar.py`—if you named yours differently, just update the following code).

The next thing we need is for the dataset to be on the virtual machine. The easiest way to do this is to go to the virtual machine and type:

```
$ wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

This will download the dataset. Once that has downloaded, you can extract the data to the Data folder by first creating that folder and then unzipping the data there:

```
$ mkdir Data

$ tar -zxf cifar-10-python.tar.gz -C Data
```

Finally, we can run our example with the following:

```
$ python3 chapter11cifar.py
```

The first thing you'll notice is a drastic speedup. On my home computer, each epoch took over 100 seconds to run. On the GPU-enabled virtual machine, each epoch takes just 16 seconds! If we tried running 100 epochs on my computer, it would take nearly three hours, compared to just 26 minutes on the virtual machine.

This drastic speedup makes trialing different models much faster. Often with trialing machine learning algorithms, the computational complexity of a single algorithm doesn't matter too much. An algorithm might take a few seconds, minutes, or hours to run. If you are only running one model, it is unlikely that this training time will matter too much—especially as prediction, as with most machine learning algorithms, is quite quick and that is where a machine learning model is mostly used.

However, when you have many parameters to run, you will suddenly need to train thousands of models with slightly different parameters—suddenly, these speed increases matter much more.

After 100 epochs of training, taking a whole 26 minutes, you will get a printout of the final result:

```
0.8497
```

Not too bad! We can increase the number of epochs of training to improve this further or we might try changing the parameters instead; perhaps, more hidden nodes, more convolution layers, or an additional dense layer. There are other types of layers in Keras that could be tried too; although generally, convolution layers are better for vision.

Summary

In this chapter, we looked at using deep neural networks, specifically convolution networks, in order to perform computer vision. We did this through the Keras package, which uses Tensorflow or Theano as its computation backend. The networks were relatively easy to build with Kera's helper functions.

The convolution networks were designed for computer vision, so it shouldn't be a surprise that the result was quite accurate. The final result shows that computer vision is indeed an effective application using today's algorithms and computational power.

We also used a GPU-enabled virtual machine to drastically speed up the process, by a factor of almost 10 for my machine. If you need extra power to run some of these algorithms, virtual machines by cloud providers can be an effective way to do this (usually for less than a dollar per hour)—just remember to turn them off when you are done!

To extend the work in this chapter, try play with the structure of the network to increase the accuracy further than what we obtained here. Another method that can be used to improve the accuracy is to create more data, either by taking your own pictures (slow) or by modifying the existing ones (much faster). To do the modification, you can flip images upside down, rotate, shear and so on. Keras has a function for doing this that is quite useful. See the documentation at `https://keras.io/preprocessing/image/`

Another area worth investigating is variations in neural network structure, more nodes, fewer nodes, more layers and so on. Also experiment with different activation types, different layer types and different combinations.

This chapter's focus was on a very complex algorithm. Convolution networks take a long time to train and have many parameters to train. Ultimately, the size of the data was small in comparison; although it was a large dataset, we can load it all in memory without even using sparse matrices. In the next chapter, we go for a much simpler algorithm, but a much, much larger dataset that can't fit in memory. This is the basis of Big Data and it underpins applications of data mining in many large industries such as mining and social networks.

12
Working with Big Data

The amount of data is increasing at an exponential rate. Today's systems are generating and recording information on customer behavior, distributed systems, network analysis, sensors, and many, many more sources. While the current big trend of mobile data is pushing the current growth, the next big thing—**the Internet of Things (IoT)**—is going to further increase the rate of growth.

What this means for data mining is a new way of thinking. Complex algorithms with high runtimes need to be improved or discarded, while simpler algorithms that can deal with more samples are becoming more popular to use. As an example, while support vector machines are great classifiers, some variants are difficult to use on very large datasets. In contrast, simpler algorithms such as logistic regression can manage more easily in these scenarios.

This complexity versus distribution issue is just one of the reasons why deep neural networks (DNNs) have become so popular. You can create very complex models using DNNs, but you can also *distribute* the workload for training them across many computers quite easily.

In this chapter, we will investigate the following:

- Big data challenges and applications
- The MapReduce paradigm
- Hadoop MapReduce
- mrjob, a Python library to run MapReduce programs on Amazon's AWS infrastructure

Big data

What makes big data different? Most big data proponents talk about the four Vs of big data:

- **Volume**: The amount of data that we generate and store is growing at an increasing rate, and predictions of the future generally only suggest further increases. Today's multi-gigabyte-sized hard drives will turn into exabyte-sized drives in a few years, and network throughput traffic will be increasing as well. The signal-to-noise ratio can be quite difficult, with important data being lost in the mountain of non-important data.
- **Velocity**: While related to volume, the velocity of data is increasing too. Modern cars have hundreds of sensors that stream data into their computers, and information from these sensors needs to be analyzed at a sub-second level to operate the car. It isn't just a case of finding answers in the volume of data; those answers often need to come quickly. In some cases, we also simply do not have enough disk space to store data, meaning we also need to make decisions on what data to keep for later analysis.
- **Variety**: Nice datasets with clearly defined columns are only a small fraction of the datasets that we have these days. Consider a social media post that may have text, photos, user mentions, likes, comments, videos, geographic information, and other fields. Simply ignoring parts of this data that don't fit your model will lead to a loss of information, but integrating that information itself can be very difficult.
- **Veracity**: With this increase in the amount of data, it can be hard to determine whether the data is being correctly collected—whether it is outdated, noisy, contains outliers—or generally whether it is useful at all. Being able to trust the data is hard when a human can't reliably verify it. External datasets are being increasingly merged into internal ones too, giving rise to more troubles relating to the veracity of data.

These main four Vs (others have proposed additional Vs) outline why big data is different from just *lots of data*. At these scales, the engineering problem of working with data is often more difficult—let alone the analysis. While there are lots of snake oil salesmen that overstate a particular product's ability to analyze big data, it is hard to deny the engineering challenges and the potential of big data analytics.

The algorithms we have used so far in the book load the dataset into memory and then work on the in-memory version. This gives a large benefit in terms of speed of computation (because using computer memory is faster than using hard drives), as it is much faster to compute on in-memory data than having to load a sample before we use it. In addition, in-memory data allows us to iterate over the data many times, thereby improving our machine learning model.

In big data, we can't load our data into memory. In many ways, this is a good definition for whether a problem is big data or not—if the data can fit in the memory on your computer, you aren't dealing with a big data problem.

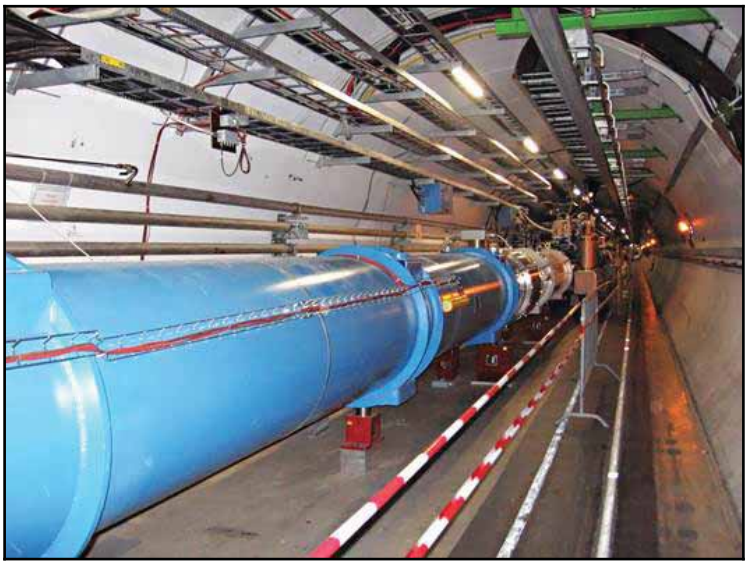
When looking at the data you create, such as log data from your company's internal applications, it might be tempting to simply throw it all into a file, unstructured, and use big-data concepts later to analyze it. It is best not to do this; instead, you should use structured formats for your own datasets. The reason is that the four Vs we just outlined are actually *problems* that need to be solved to perform data analysis, not goals to strive for!

Applications of big data

There are many use cases for big data, in public and private sectors.

The most common experience people have using a big-data-based system is in Internet search, such as Google. To run these systems, a search needs to be carried out over billions of websites in a fraction of a second. Doing a basic text-based search would be inadequate to deal with such a problem. Simply storing the text of all those websites is a large problem. In order to deal with queries, new data structures and data mining methods need to be created and implemented specifically for this application.

Big data is also used in many other scientific experiments such as the Large Hadron Collider, part of which is pictured next. It stretches over 27 kilometers and contains 150 million sensors monitoring hundreds of millions of particle collisions per second. The data from this experiment is massive, with 25 petabytes created daily, after a filtering process (if filtering were not used, there would be 150 million petabytes per year). Analysis on data this big has led to amazing insights about our universe, but it has been a significant engineering and analytics challenge.



Governments are increasingly using big data too, to track populations, businesses, and other aspects related to their country. Tracking millions of people and billions of interactions (such as business transactions or health spending) leads to a need for big data analytics in many government organizations.

Traffic management is a particular focus of many governments around the world, who are tracking traffic using millions of sensors to determine which roads are the most congested and predicting the impact of new roads on traffic levels. These management systems will link with data from autonomous cars in the near future, leading to even more data about traffic conditions in real time. Cities that make use of this data will find that their traffic flows more freely.

Large retail organizations are using big data to improve customer experience and reduce costs. This involves predicting customer demand in order to have the correct level of inventory, upselling customers with products they may like to purchase, and tracking transactions to look for trends, patterns, and potential frauds. Companies that automatically create great predictions can have higher sales at lower costs.

Other large businesses are also leveraging big data to automate aspects of their business and improve their offering. This includes leveraging analytics to predict future trends in their sector and tracking external competitors. Large businesses also use analytics to manage their own employees—tracking employees to look for signs that an employee may leave the company, in order to intervene before they do.

The information security sector is also leveraging big data in order to look for malware infections in large networks, by monitoring network traffic. This can include looking for odd traffic patterns, evidence of malware spreading, and other oddities. Advanced Persistent Threats (APTs) is another problem, where a motivated attacker will hide their code within a large network to steal information or cause damage over a long period of time. Finding APTs is often a case of forensically examining many computers, a task which simply takes too long for a human to effectively perform themselves. Analytics helps automate and analyze these forensic images to find infections.

Big data is being used in an increasing number of sectors and applications, and this trend is likely to only continue.

MapReduce

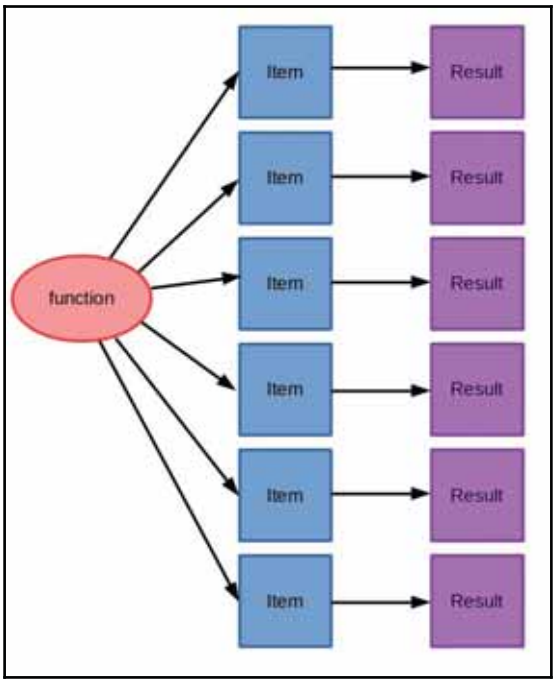
There are a number of concepts to perform data mining and general computation on big data. One of the most popular is the MapReduce model, which can be used for general computation on arbitrarily large datasets.

MapReduce originates from Google, where it was developed with distributed computing in mind. It also introduces fault tolerance and scalability improvements. The *original* research for MapReduce was published in 2004, and since then there have been thousands of projects, implementations, and applications using it.

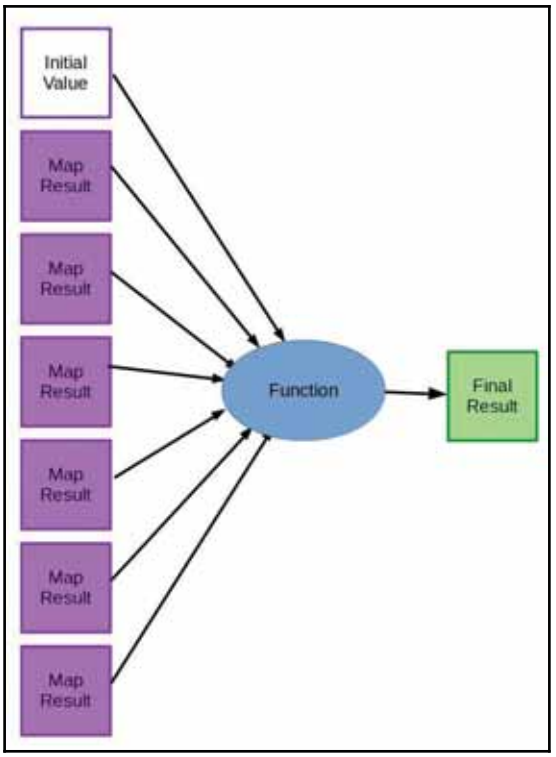
While the concept is similar to many previous concepts, MapReduce has become a staple in big data analytics.

There are two major stages in a MapReduce job.

1. The first is Map, by which we take a function and a list of items, and apply that function to each item. Put another way, we take each item as the input to the function and store the result of that function call:



2. The second step is Reduce, where we take the results from the map step and combine them using a function. For statistics, this could be as simple as adding all the numbers together. The reduce function in this scenario is an add function, which would take the previous sum, and add the new result:



After these two steps, we will have transformed our data and reduced it to a final result.

MapReduce jobs can have many iterations, some of which are only Map jobs, some only Reduce jobs and some iterations with both a Map and Reduce step. Let us now have a look at some more tangible examples, first using built-in python functions and then using a specific tool for MapReduce jobs.

The intuition behind MapReduce

MapReduce has two main steps: the `Map` step and the `Reduce` step. These are built on the functional programming concepts of mapping a function to a list and reducing the result. To explain the concept, we will develop code that will iterate over a list of lists and produce the sum of all numbers in those lists.

There are also `shuffle` and `combine` steps in the MapReduce paradigm, which we will see later.

To start with, the Map step takes a function and applies it to each element in a list. The returned result is a list of the same size, with the results of the function applied to each element.

To open a new Jupyter Notebook, start by creating a list of lists with numbers in each sublist:

```
a = [[1,2,1], [3,2], [4,9,1,0,2]]
```

Next, we can perform a `map` using the sum function. This step will apply the **sum** function to each element of *a*:

```
sums = map(sum, a)
```

While `sums` is a generator (the actual value isn't computed until we ask for it), the preceding step is approximately equal to the following code:

```
sums = []
for sublist in a:
    results = sum(sublist)
    sums.append(results)
```

The `reduce` step is a little more complicated. It involves applying a function to each element of the returned result, to some starting value. We start with an initial value and then apply a given function to that initial value and the first value. We then apply the given function to the result and the next value, and so on

We start by creating a function that takes two numbers and adds them together.

```
def add(a, b):
    return a + b
```

We then perform the reduce. The signature of `reduce` is: `reduce(function, sequence, initial)`, where the function is applied at each step to the sequence. In the first step, the initial value is used as the first value rather than the first element of the list:

```
from functools import reduce
print(reduce(add, sums, 0))
```

The result, 25, is the sum of each of the values in the sums list and is consequently the sum of each of the elements in the original array.

The preceding code is similar to the following:

```
initial = 0
current_result = initial
for element in sums:
    current_result = add(current_result, element)
```

In this simple example, our code would be greatly simplified if it didn't use the MapReduce paradigm, but the real gains come from distributing the computation. For instance, if we have a million sublists and each of those sublists contains a million elements, we can distribute this computation over many computers.

In order to do this, we distribute the `map` step by segmenting out data. For each of the elements in our list, we send it, along with a description of our function, to a computer. This computer then returns the result to our main computer (the master).

The master then sends the result to a computer for the `reduce` step. In our example of a million sublists, we would send a million jobs to different computers (the same computer may be reused after it completes our first job). The returned result would be just a single list of a million numbers, which we then compute the sum of.

The result is that no computer ever needed to store more than a million numbers, despite our original data having a trillion numbers in it.

A word count example

Any *actual* implementation of MapReduce is a little more complex than just using a `map` and `reduce` step. Both steps are invoked using keys, which allows for the separation of data and tracking of values.

The map function takes a key-value pair and returns a list of *key/value* pairs. The keys for the input and output don't necessarily relate to each other.

For example, for a MapReduce program that performs a word count, the input key might be a sample document's ID value, while the output key would be a given word. The input value would be the text of the document and the output value would be the frequency of each word. We split the document to get the words, and then yield each of the word, count pairs. The word here is the key, with the count being the value in MapReduce terms:

```
from collections import defaultdict
def map_word_count(document_id, document):
    counts = defaultdict(int)
    for word in document.split():
        counts[word] += 1
    for word in counts:
        yield (word, counts[word])
```

Have a really, really big dataset? You can just do `yield (word, 1)` when you come across a new word and then combine the ones in the shuffle step rather than count within the map step. Where you place it depends on your dataset size, per-document size, network capacity, and a whole range of factors. Big data is a big engineering problem and to get the maximum performance out of a system, you'll need to model how data will flow throughout the algorithm.

By using the word as the key, we can then perform a shuffle step, which groups all the values for each key:

```
def shuffle_words(results):
    records = defaultdict(list)
    for results in results_generators:
        for word, count in results:
            records[word].append(count)
    for word in records:
        yield (word, records[word])
```

The final step is the reduce step, which takes a key-value pair (the value, in this case, is always a list) and produces a key-value pair as a result. In our example, the key is the word, the input list is the list of counts produced in the shuffle step, and the output value is the sum of the counts:

```
def reduce_counts(word, list_of_counts):
    return (word, sum(list_of_counts))
```

To see this in action, we can use the 20 newsgroups dataset, which is provided in scikit-learn. This dataset is not big data, but we can see the concepts in action here:

```
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(subset='train')
documents = dataset.data
```

We then apply our map step. We use enumerate here to automatically generate document IDs for us. While they aren't important in this application, these keys are important in other applications:

```
map_results = map(map_word_count, enumerate(documents))
```

The actual result here is just a generator; no actual counts have been produced. That said, it is a generator that emits **(word, count)** pairs.

Next, we perform the shuffle step to sort these word counts:

```
shuffle_results = shuffle_words(map_results)
```

This, in essence, is a MapReduce job; however, it is only running on a single thread, meaning we aren't getting any benefit from the MapReduce data format. In the next section, we will start using Hadoop, an open source provider of MapReduce, to start getting the benefits of this type of paradigm.

Hadoop MapReduce

Hadoop is a set of open source tools from Apache that includes an implementation of MapReduce. In many cases, it is the de-facto implementation used by many. The project is managed by the Apache group (who are responsible for the famous web server of the same name).

The Hadoop ecosystem is quite complex, with a large number of tools. The main component we will use is Hadoop MapReduce. Other tools for working with big data that are included in Hadoop are as follows:

- **Hadoop Distributed File System (HDFS)**: This is a file system that can store files over many computers, with the goal of being robust against hardware failure while providing high bandwidth.
- **YARN**: This is a method for scheduling applications and managing clusters of computers.

- **Pig:** This is a higher level programming language for MapReduce. Hadoop MapReduce is implemented in Java, and Pig sits on top of the Java implementation, allowing you to write programs in other languages—including Python.
- **Hive:** This is for managing data warehouses and performing queries.
- **HBase**: This is an implementation of Google's BigTable, a distributed database.

These tools all solve different issues that come up when doing big data experiments, including data analytics.

There are also non-Hadoop-based implementations of MapReduce, as well as other projects with similar goals. In addition, many cloud providers have MapReduce-based systems.

Applying MapReduce

In this application, we will look at predicting the gender of a writer based on their use of different words. We will use a Naive Bayes method for this, trained in MapReduce. The final model doesn't need MapReduce, although we can use the Map step to do so—that is, run the prediction model on each document in a list. This is a common Map operation for data mining in MapReduce, with the reduce step simply organizing the list of predictions so they can be tracked back to the original document.

We will be using Amazon's infrastructure to run our application, allowing us to leverage their computing resources.

Getting the data

The data we are going to use is a set of blog posts that are labeled for age, gender, industry (that is, work) and, funnily enough, star sign. This data was collected from `http://blogger.com`in August 2004 and has over 140 million words in more than 600,000 posts. Each blog is *probably* written by just one person, with some work put into verifying this (although, we can never be really sure). Posts are also matched with the date of posting, making this a very rich dataset.

To get the data, go to `http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm` and click on **Download Corpus**. From there, unzip the file to a directory on your computer.

The dataset is organized with a single blog to a file, with the filename giving the classes. For instance, one of the filenames is as follows:

```
1005545.male.25.Engineering.Sagittarius.xml
```

The filename is separated by periods, and the fields are as follows:

- **Blogger ID**: This a simple ID value to organize the identities.
- **Gender**: This is either male or female, and all the blogs are identified as one of these two options (no other options are included in this dataset).
- **Age**: The exact ages are given, but some gaps are deliberately present. Ages present are in the (inclusive) ranges of 13-17, 23-27, and 33-48. The reason for the gaps is to allow for splitting the blogs into age ranges with gaps, as it would be quite difficult to separate an 18-year old's writing from a 19-year-old, and it is possible that the age itself is a little outdated itself and would need to be updated to 19 anyway.
- **Industry**: In one of 40 different industries including science, engineering, arts, and real estate. Also, included is indUnk, for an unknown industry.
- **Star Sign**: This is one of the 12 astrological star signs.

All values are self-reported, meaning there may be errors or inconsistencies with labeling, but are assumed to be mostly reliable—people had the option of not setting values if they wanted to preserve their privacy in those ways.

A single file is in a pseudo-XML format, containing a `<Blog>` tag and then a sequence of `<post>` tags. Each of the `<post>` tag is proceeded by a `<date>` tag as well. While we can parse this as XML, it is much simpler to parse it on a line-by-line basis as the files are not exactly well-formed XML, with some errors (mostly encoding problems). To read the posts in the file, we can use a loop to iterate over the lines.

We set a test filename so we can see this in action:

```
import os
filename = os.path.join(os.path.expanduser("~"), "Data", "blogs",
"1005545.male.25.Engineering.Sagittarius.xml")
```

First, we create a list that will let us store each of the posts:

```
all_posts = []
```

Then, we open the file to read:

```
with open(filename) as inf:
    post_start = False
    post = []
    for line in inf:
        line = line.strip()
        if line == "<post>":
            # Found a new post
            post_start = True
        elif line == "</post>":
            # End of the current post, append to our list of posts and
start a new one
            post_start = False
            all_posts.append("n".join(post))
            post = []
        elif post_start:
            # In a current post, add the line to the text of the post
            post.append(line)
```

If we aren't in a current post, we simply ignore the line.

We can then grab the text of each post:

```
print(all_posts[0])
```

We can also find out how many posts this author created:

```
print(len(all_posts))
```

Naive Bayes prediction

We are now going to implement the Naive Bayes algorithm using mrjob, allowing it to process our dataset. Technically our version will be a reduced version of most Naive Bayes' implementations, without many of the features that you would expect like smoothing small values.

The mrjob package

The **mrjob** package allows us to create MapReduce jobs that can easily be computed on Amazon's infrastructure. While mrjob sounds like a sedulous addition to the Mr. Men series of children's books, it stands for **Map Reduce Job**.

You can install mrjob using the following: `pip install mrjob`
I had to install the **filechunkio** package separately using `conda install -c conda-forge filechunkio`, but this will depend on your system setup. There are other Anaconda channels for installing mrjob, check them with:
`anaconda search -t conda mrjob`

In essence, **mrjob** provides the standard functionality that most MapReduce jobs need. Its most amazing feature is that you can write the same code, test on your local machine (without heavy infrastructure like Hadoop), and then push to Amazon's EMR service or another Hadoop server.

This makes testing the code significantly easier, although it can't magically make a big problem small— any local testing uses a subset of the dataset, rather than the whole, big dataset. Instead, mrjob gives you a framework that you can test with a small problem and have more confidence that the solution will scale to a larger problem, distributed on different systems.

Extracting the blog posts

We are first going to create a MapReduce program that will extract each of the posts from each blog file and store them as separate entries. As we are interested in the gender of the author of the posts, we will extract that too and store it with the post.

We can't do this in a Jupyter Notebook, so instead open a Python IDE for development. If you don't have a Python IDE you can use a text editor. I recommend PyCharm, although it has a larger learning curve and it is probably a bit heavy for just this chapter's code.

At the very least, I recommend using an IDE that has syntax highlighting and basic completion of variable names (that last one helps find typos in your code easily.

If you still can't find an IDE you like, you can write the code in an IPython Notebook and then click on **File| Download As| Python**. Save this file to a directory and run it as we outlined in `Chapter 11`, *Classifying Objects in Images using Deep Learning*.

To do this, we will need the `os` and `re` libraries as we will be obtaining environment variables and we will also use a regular expression for word separation:

```
import os
import re
```

We then import the MRJob class, which we will inherit from our MapReduce job:

```
from mrjob.job import MRJob
```

We then create a new class that subclasses MRJob. We will use a similar loop, as before, to extract blog posts from the file. The mapping function we will define next will work off each line, meaning we have to track different posts outside of the mapping function. For this reason, we make `post_start` and post class variables, rather than variables inside the function. We then define our mapper function—this takes a line from a file as input and yields blog posts. The lines are guaranteed to be ordered from the same per-job file. This allows us to use the above class variables to record current post data:

```
class ExtractPosts(MRJob):
    post_start = False
    post = []

    def mapper(self, key, line):
        filename = os.environ["map_input_file"]
        # split the filename to get the gender (which is the second token)
        gender = filename.split(".")[1]
        line = line.strip()
        if line == "<post>":
            self.post_start = True
        elif line == "</post>":
            self.post_start = False
            yield gender, repr("n".join(self.post))
            self.post = []
        elif self.post_start:
            self.post.append(line)
```

Rather than storing the posts in a list, as we did earlier, we yield them. This allows mrjob to track the output. We yield both the gender and the post so that we can keep a record of which gender each record matches. The rest of this function is defined in the same way as our loop above.

Finally, outside the function and class, we set the script to run this MapReduce job when it is called from the command line:

```
if __name__ == '__main__':
    ExtractPosts.run()
```

Now, we can run this MapReduce job using the following shell command.

```
$ python extract_posts.py <your_data_folder>/blogs/51*
  --output-dir=<your_data_folder>/blogposts --no-output
```

Just a reminder that you don't need to enter the **$** on the above line - this just indicates this is a command run from the command line, and not in a Jupyter Notebook.

The first parameter, `<your_data_folder>/blogs/51*` (just remember to change `<your_data_folder>` to the full path to your data folder), obtains a sample of the data (all files starting with 51, which is only 11 documents). We then set the output directory to a new folder, which we put in the data folder, and specify not to output the streamed data. Without the last option, the output data is shown to the command line when we run it—which isn't very helpful to us and slows down the computer quite a lot.

Run the script, and quite quickly each of the blog posts will be extracted and stored in our output folder. This script only ran on a single thread on the local computer so we didn't get a speedup at all, but we know the code runs.

We can now look in the output folder for the results. A bunch of files are created and each file contains each blog post on a separate line, preceded by the gender of the author of the blog.

Training Naive Bayes

Now that we have extracted the blog posts, we can train our Naive Bayes model on them. The intuition is that we record the probability of a word being written by a particular gender, and record these values in our model. To classify a new sample, we would multiply the probabilities and find the most likely gender.

The aim of this code is to output a file that lists each word in the corpus, along with the frequencies of that word for each gender. The output file will look something like this:

```
"'ailleurs" {"female": 0.003205128205128205}
"'air" {"female": 0.003205128205128205}
"'an" {"male": 0.0030581039755351682, "female": 0.004273504273504274}
"'angoisse" {"female": 0.003205128205128205}
"'apprendra" {"male": 0.0013047113868622459, "female":
0.0014172668603481887}
"'attendent" {"female": 0.00641025641025641}
"'autistic" {"male": 0.002150537634408602}
```

```
"'auto" {"female": 0.003205128205128205}
"'avais" {"female": 0.00641025641025641}
"'avait" {"female": 0.004273504273504274}
"'behind" {"male": 0.0024390243902439024}
"'bout" {"female": 0.002034152292059272}
```

The first value is the word and the second is a dictionary mapping the genders to the frequency of that word in that gender's writings.

Open a new file in your Python IDE or text editor. We will again need the `os` and `re` libraries, as well as `NumPy` and `MRJob` from `mrjob`. We also need `itemgetter`, as we will be sorting a dictionary:

```
import os
import re
import numpy as np
from mrjob.job import MRJob
from operator import itemgetter
```

We will also need `MRStep`, which outlines a step in a MapReduce job. Our previous job only had a single step, which is defined as a mapping function and then as a reducing function. This job will have multiple steps where we Map, Reduce, and then Map and Reduce again. The intuition is the same as the pipelines we used in earlier chapters, where the output of one step is the input to the next step:

```
from mrjob.step import MRStep
```

We then create our word search regular expression and compile it, allowing us to find word boundaries. This type of regular expression is much more powerful than the simple split we used in some previous chapters, but if you are looking for a more accurate word splitter, I recommend using NLTK or Spacey as we did in `Chapter 6`, *Social Media Insight using Naive Bayes*:

```
word_search_re = re.compile(r"[w']+")
```

We define a new class for our training. I'll first provide the whole class as one code block, and then we will come back to each section to review what it does:

```
class NaiveBayesTrainer(MRJob):

    def steps(self):
    return [
            MRStep(mapper=self.extract_words_mapping,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.compare_words_reducer),
    ]
```

```
    def extract_words_mapping(self, key, value):
        tokens = value.split()
        gender = eval(tokens[0])
        blog_post = eval(" ".join(tokens[1:]))
        all_words = word_search_re.findall(blog_post)
        all_words = [word.lower() for word in all_words]
        for word in all_words:
            # Occurence probability
            yield (gender, word), 1. / len(all_words)

    def reducer_count_words(self, key, counts):
        s = sum(counts)
        gender, word = key #.split(":")
        yield word, (gender, s)

    def compare_words_reducer(self, word, values):
        per_gender = {}
        for value in values:
            gender, s = value
            per_gender[gender] = s
            yield word, per_gender

    def ratio_mapper(self, word, value):
        counts = dict(value)
        sum_of_counts = float(np.mean(counts.values()))
        maximum_score = max(counts.items(), key=itemgetter(1))
        current_ratio = maximum_score[1] / sum_of_counts
        yield None, (word, sum_of_counts, value)

    def sorter_reducer(self, key, values):
        ranked_list = sorted(values, key=itemgetter(1), reverse=True)
        n_printed = 0
        for word, sum_of_counts, scores in ranked_list:
            if n_printed < 20:
                print((n_printed + 1), word, scores)
            n_printed += 1
        yield word, dict(scores)
```

Let's have a look at the sections of this code, one step at a time:

```
class NaiveBayesTrainer(MRJob):
```

We define the steps of our MapReduce job. There are two steps:

The first step will extract the word occurrence probabilities. The second step will compare the two genders and output the probabilities for each to our output file. In each MRStep, we define the mapper and reducer functions, which are class functions in this NaiveBayesTrainer class (we will write those functions next):

```
def steps(self):
    return [
        MRStep(mapper=self.extract_words_mapping,
               reducer=self.reducer_count_words),
        MRStep(reducer=self.compare_words_reducer),
    ]
```

The first function is the mapper function for the first step. The goal of this function is to take each blog post, get all the words in that post, and then note the occurrence. We want the frequencies of the words, so we will return `1 / len(all_words)`, which allows us to later sum the values for frequencies. The computation here isn't exactly correct—we need to also normalize for the number of documents. In this dataset, however, the class sizes are the same, so we can conveniently ignore this with little impact on our final version.

We also output the gender of the post's author, as we will need that later:

```
def extract_words_mapping(self, key, value):
    tokens = value.split()
    gender = eval(tokens[0])
    blog_post = eval(" ".join(tokens[1:]))
    all_words = word_search_re.findall(blog_post)
    all_words = [word.lower() for word in all_words]
    for word in all_words:
        # Occurence probability
        yield (gender, word), 1. / len(all_words)
```

We used `eval` in the preceding code to simplify the parsing of the blog posts from the file, for this example. This is not recommended. Instead, use a format such as JSON to properly store and parse the data from the files. A malicious user with access to the dataset can insert code into these tokens and have that code run on your server.

In the reducer for the first step, we sum the frequencies for each gender and word pair. We also change the key to be the word, rather than the combination, as this allows us to search by word when we use the final trained model (although, we still need to output the gender for later use);

```
def reducer_count_words(self, key, counts):
    s = sum(counts)
```

```
            gender, word = key #.split(":")
            yield word, (gender, s)
```

The final step doesn't need a mapper function, which is why we didn't add one. The data will pass straight through as a type of identity mapper. The reducer, however, will combine frequencies for each gender under the given word and then output the word and frequency dictionary.

This gives us the information we needed for our Naive Bayes implementation:

```
        def compare_words_reducer(self, word, values):
            per_gender = {}
            for value in values:
                gender, s = value
                per_gender[gender] = s
                yield word, per_gender
```

Finally, we set the code to run this model when the file is run as a script. We will need to add this code to the file:

```
if __name__ == '__main__':
 NaiveBayesTrainer.run()
```

We can then run this script. The input to this script is the output of the previous post-extractor script (we can actually have them as different steps in the same MapReduce job if you are so inclined);

```
$ python nb_train.py <your_data_folder>/blogposts/
 --output-dir=<your_data_folder>/models/ --no-output
```

The output directory is a folder that will store a file containing the output from this MapReduce job, which will be the probabilities we need to run our Naive Bayes classifier.

Putting it all together

We can now actually run the Naive Bayes classifier using these probabilities. We will do this in a Jupyter Notebook, although this processing itself can be transferred to a mrjob package to be performed at scale.

First, take a look at the `models` folder that was specified in the last MapReduce job. If the output was more than one file, we can merge the files by just appending them to each other using a command line function from within the `models` directory:

```
cat * > model.txt
```

If you do this, you'll need to update the following code with `model.txt` as the model filename.

Back to our Notebook, we first import some standard imports we need for our script:

```
import os
import re
import numpy as np
from collections import defaultdict
from operator import itemgetter
```

We again redefine our word search regular expression—if you were doing this in a real application, I recommend centralizing the functionality. It is important that words are extracted in the same way for both training and testing:

```
word_search_re = re.compile(r"[w']+")
```

Next, we create the function that loads our model from a given filename. The model parameters will take the form of a dictionary of dictionaries, where the first key is a word, and the inner dictionary maps each gender to a probability. We use `defaultdicts`, which will return zero if a value isn't present;

```
def load_model(model_filename):
    model = defaultdict(lambda: defaultdict(float))
    with open(model_filename) as inf:
        for line in inf:
            word, values = line.split(maxsplit=1)
            word = eval(word)
            values = eval(values)
            model[word] = values
    return model
```

The line is split into two sections, separated by whitespace. The first is the word itself and the second is a dictionary of probabilities. For each, we run `eval` on them to get the actual value, which was stored using `repr` in the previous code.

Next, we load our actual model. You may need to change the model filename—it will be in the output dir of the last MapReduce job;

```
model_filename = os.path.join(os.path.expanduser("~"), "models",
"part-00000")
model = load_model(model_filename)
```

As an example, we can see the difference in usage of the word *i* (all words are turned into lowercase in the MapReduce jobs) between males and females:

```
model["i"]["male"], model["i"]["female"]
```

Next, we create a function that can use this model for prediction. We won't use the scikit-learn interface for this example, and just create a function instead. Our function takes the model and a document as the parameters and returns the most likely gender:

```
def nb_predict(model, document):
    probabilities = defaultdict(lambda : 1)
    words = word_search_re.findall(document)
    for word in set(words):
        probabilities["male"] += np.log(model[word].get("male", 1e-15))
        probabilities["female"] += np.log(model[word].get("female", 1e-15))
        most_likely_genders = sorted(probabilities.items(),
key=itemgetter(1), reverse=True)
    return most_likely_genders[0][0]
```

It is important to note that we used `np.log` to compute the probabilities. Probabilities in Naive Bayes models are often quite small. Multiplying small values, which is necessary for many statistical values, can lead to an underflow error where the computer's precision isn't good enough and just makes the whole value 0. In this case, it would cause the likelihoods for both genders to be zero, leading to incorrect predictions.

To get around this, we use log probabilities. For two values a and b, $log(a \times b)$ is equal to $log(a) + log(b)$. The log of a small probability is a negative value, but a relatively large one. For instance, log(0.00001) is about -11.5. This means that rather than multiplying actual probabilities and risking an underflow error, we can sum the log probabilities and compare the values in the same way (higher numbers still indicate a higher likelihood).

If you want to obtain probabilities back from the log probabilities, make sure to undo the log operation by using e to the power of the value you are interested in. To revert -11.5 into the probability, take $e^{-11.5}$, which equals 0.00001 (approximately).

One problem with using log probabilities is that they don't handle zero values well (although, neither does multiplying by zero probabilities). This is due to the fact that log(0) is undefined. In some implementations of Naive Bayes, a 1 is added to all counts to get rid of this, but there are other ways to address this. This is a simple form of smoothing of the values. In our code, we just return a very small value if the word hasn't been seen for our given gender.

Adding one to all counts above is a form of smoothing. Another option is to initialise to a very small value, such as 10^{-16} - as long as its not exactly 0!

Back to our prediction function, we can test this by copying a post from our dataset:

```
new_post = """ Every day should be a half day. Took the afternoon off to
hit the dentist, and while I was out I managed to get my oil changed, too.
Remember that business with my car dealership this winter? Well, consider
this the epilogue. The friendly fellas at the Valvoline Instant Oil Change
on Snelling were nice enough to notice that my dipstick was broken, and the
metal piece was too far down in its little dipstick tube to pull out. Looks
like I'm going to need a magnet. Damn you, Kline Nissan, daaaaaaammmnnn
yooouuuu.... Today I let my boss know that I've submitted my Corps
application. The news has been greeted by everyone in the company with a
level of enthusiasm that really floors me. The back deck has finally been
cleared off by the construction company working on the place. This company,
for anyone who's interested, consists mainly of one guy who spends his days
cursing at his crew of Spanish-speaking laborers. Construction of my deck
began around the time Nixon was getting out of office.
"""
```

We then predict with the following code:

```
nb_predict(model, new_post)
```

The resulting prediction, *male*, is correct for this example. Of course, we never test a model on a single sample. We used the file starting with 51 for training this model. It wasn't many samples, so we can't expect too high of an accuracy.

The first thing we should do is train on more samples. We will test on any file that starts with a 6 or 7 and train on the rest of the files.

In the command line and in your data folder (`cd <your_data_folder>`), where the blogs folder exists, create a copy of the **blogs** folder into a new folder.

Make a folder for our training set:

```
mkdir blogs_train
```

Move any file starting with a 6 or 7 into the test set, from the train set:

```
cp blogs/4* blogs_train/
cp blogs/8* blogs_train/
```

Then, make a folder for our test set:

```
mkdir blogs_test
```

Move any file starting with a 6 or 7 into the test set, from the train set:

```
cp blogs/6* blogs_test/
cp blogs/7* blogs_test/
```

We will rerun the blog extraction on all files in the training set. However, this is a large computation that is better suited to cloud infrastructure than our system. For this reason, we will now move the parsing job to Amazon's infrastructure.

Run the following on the command line, as you did before. The only difference is that we train on a different folder of input files. Before you run the following code, delete all files in the blog posts and models folders:

```
$ python extract_posts.py ~/Data/blogs_train --output-
dir=/home/bob/Data/blogposts_train --no-output
```

Next up comes the training of our Naive Bayes model. The code here will take quite a bit longer to run. Many, many hours. You may want to skip running this locally, unless you have a really powerful system! If you do want to skip, head to the next section.

```
$ python nb_train.py ~/Data/blogposts_train/ --output-dir=/home/bob/models/
--no-output
```

We will test on any blog file in our test set. To get the files, we need to extract them. We will use the `extract_posts.py` MapReduce job, but store the files in a separate folder:

```
python extract_posts.py ~/Data/blogs_test --output-
dir=/home/bob/Data/blogposts_test --no-output
```

Back in the Jupyter Notebook, we list all the outputted testing files:

```
testing_folder = os.path.join(os.path.expanduser("~"), "Data",
"blogposts_testing")
testing_filenames = []
for filename in os.listdir(testing_folder):
    testing_filenames.append(os.path.join(testing_folder, filename))
```

For each of these files, we extract the gender and document and then call the predict function. We do this in a generator, as there are a lot of documents, and we don't want to use too much memory. The generator yields the actual gender and the predicted gender:

```
def nb_predict_many(model, input_filename):
    with open(input_filename) as inf: # remove leading and trailing
whitespace
    for line in inf:
        tokens = line.split()
        actual_gender = eval(tokens[0])
        blog_post = eval(" ".join(tokens[1:]))
        yield actual_gender, nb_predict(model, blog_post)
```

We then record the predictions and actual genders across our entire dataset. Our predictions here are either male or female. In order to use the `f1_score` function from scikit-learn, we need to turn these into ones and zeroes. In order to do that, we record a 0 if the gender is male and 1 if it is female. To do this, we use a Boolean test, seeing if the gender is female. We then convert these Boolean values to `int` using NumPy:

```
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(model,
testing_filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
    y_true = np.array(y_true, dtype='int')
    y_pred = np.array(y_pred, dtype='int')
```

Now, we test the quality of this result using the F1 score in scikit-learn:

```
from sklearn.metrics import f1_score
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

The result of 0.78 is quite reasonable. We can probably improve this by using more data, but to do that, we need to move to a more powerful infrastructure that can handle it.

Training on Amazon's EMR infrastructure

We are going to use Amazon's **Elastic Map Reduce** (**EMR**) infrastructure to run our parsing and model building jobs.

In order to do that, we first need to create a bucket in Amazon's storage cloud. To do this, open the Amazon S3 console in your web browser by going to `http://console.aws.amazon.com/s3` and click on **Create Bucket**. Remember the name of the bucket, as we will need it later.

Right-click on the new bucket and select Properties. Then, change the permissions, granting everyone full access. This is not a good security practice in general, and I recommend that you change the access permissions after you complete this chapter. You can use advanced permissions in Amazon's services to give your script access and also protect against third parties viewing your data.

Left-click the bucket to open it and click on Create Folder. Name the folder **blogs_train**. We are going to upload our training data to this folder for processing on the cloud.

On your computer, we are going to use Amazon's AWS CLI, a command-line interface for processing on Amazon's cloud.

To install it, use the following:

```
sudo pip install awscli
```

Follow the instructions at `http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-set-up.html`to set the credentials for this program.

We now want to upload our data to our new bucket. First, we want to create our dataset, which is all the blogs not starting with a 6 or 7. There are more graceful ways to do this copy, but none are cross-platform enough to recommend. Instead, simply copy all the files and then delete the ones that start with a 6 or 7, from the training dataset:

```
cp -R ~/Data/blogs ~/Data/blogs_train_large
rm ~/Data/blogs_train_large/blogs/6*
rm ~/Data/blogs_train_large/blogs/7*
```

Next, upload the data to your Amazon S3 bucket. Note that this will take some time and use quite a lot of upload data (several hundred megabytes). For those with slower internet connections, it may be worth doing this at a location with a faster connection;

```
aws s3 cp ~/Data/blogs_train_large/ s3://ch12/blogs_train_large --recursive
--exclude "*"
--include "*.xml"
```

We are going to connect to Amazon's EMR (Elastic Map Reduce) using mrjob—it handles the whole thing for us; it only needs our credentials to do so. Follow the instructions at `https://pythonhosted.org/mrjob/guides/emr-quickstart.html`to setup mrjob with your Amazon credentials.

After this is done, we alter our mrjob run, only slightly, to run on Amazon EMR. We just tell mrjob to use **emr** using the **-r** switch and then set our s3 containers as the input and output directories. Even though this will be run on Amazon's infrastructure, it will still take quite a long time to run, as the default settings for mrjob use a single, low powered computer.

```
$ python extract_posts.py -r emr s3://ch12gender/blogs_train_large/
--output-dir=s3://ch12/blogposts_train/ --no-output
$ python nb_train.py -r emr s3://ch12/blogposts_train/ --output-
dir=s3://ch12/model/ --o-output
```

You will be charged for the usage of both S3 and EMR. This will only be a few dollars, but keep this in mind if you are going to keep running the jobs or doing other jobs on bigger datasets. I ran a very large number of jobs and was charged about $20 all up. Running just these few should be less than $4. However, you can check your balance and set up pricing alerts, by going to `https://console.aws.amazon.com/billing/home`

It isn't necessary for the **blogposts_train** and model folders to exist—they will be created by EMR. In fact, if they exist, you will get an error. If you are rerunning this, just change the names of these folders to something new, but remember to change both commands to the same names (that is, the output directory of the first command is the input directory of the second command).

If you are getting impatient, you can always stop the first job after a while and just use the training data gathered so far. I recommend leaving the job for an absolute minimum of 15 minutes and probably at least an hour. You can't stop the second job and get good results though; the second job will probably take about two to three times as long as the first job did.

If you have the ability to purchase more advanced hardware, mrjob supports the creation of clusters on Amazon's infrastructure and also the ability to use more powerful computing hardware. You can run a job on a cluster of machines by specifying the type and number at the command line. For instance, to use 16 c1.medium computers to extract the text, run the following command:

```
$ python extract_posts.py -r emr s3://chapter12/blogs_train_large/blogs/ --
output-dir=s3://chapter12/blogposts_train/ --no-output  --instance-type
c1.medium --num-core-instances 16
```

In addition, you can create clusters separately and reattach jobs to those clusters. See mrjob's documentation at `https://pythonhosted.org/mrjob/guides/emr-advanced.html`for more information on this process. Keep in mind that more advanced options become an interaction between advanced features of mrjob and advanced features of Amazon's AWS infrastrucutre, meaning you will need to investigate both technologies to get high-powered processing. Keep in mind that if you run more instances of more powerful hardware, you will be charged more in turn.

You can now go back to the s3 console and download the output model from your bucket. Saving it locally, we can go back to our Jupyter Notebook and use the new model. We reenter the code here—only the differences are highlighted, just to update to our new model:

```
ws_model_filename = os.path.join(os.path.expanduser("~"), "models",
"aws_model")
aws_model = load_model(aws_model_filename)
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(aws_model,
testing_filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

The result is better with the extra data, at 0.81.

If everything went as planned, you may want to remove the bucket from Amazon S3—you will be charged for the storage.

Summary

In this chapter, we looked at running jobs on big data. By most standards, our dataset is actually quite small—only a few hundred megabytes. Many industrial datasets are much bigger, so extra processing power is needed to perform the computation. In addition, the algorithms we used can be optimized for different tasks to further increase the scalability.

Our approach extracted word frequencies from blog posts, in order to predict the gender of the author of a document. We extracted the blogs and word frequencies using MapReduce-based projects in mrjob. With those extracted, we can then perform a Naive Bayes-esque computation to predict the gender of a new document.

We only scratched the surface of what you can do with MapReduce, and we did not even use it to its full potential for this application. To take the lessons further, convert the prediction function to a MapReduce job. That is, you train the model on MapReduce to obtain a model, and you run the model with MapReduce to get a list of predictions. Extend this by also doing your evaluation in MapReduce, with the final result coming back as simply the F1-score!

We can use the mrjob library to test locally and then automatically set up and use Amazon's EMR cloud infrastructure. You can use other cloud infrastructure or even a custom built Amazon EMR cluster to run these MapReduce jobs, but there is a bit more tinkering needed to get them running.

Next Steps...

During the course, there were lots of avenues not taken, options not presented, and subjects not fully explored. In this appendix, I've created a collection of next steps for those wishing to undertake extra learning and progress their data mining with Python.

This appendix is for learning more about data mining. Also included are some challenges to extend the work performed. Some of these will be small improvements; some will be quite a bit more work—I've made a note of those more tasks that are noticeably more difficult and involved than the others.

Getting Started with Data Mining

In this chapter following are a few avenues that reader can explore:

Scikit-learn tutorials

URL: `http://scikit-learn.org/stable/tutorial/index.html`

Included in the scikit-learn documentation is a series of tutorials on data mining. The tutorials range from basic introductions to toy datasets, all the way through to comprehensive tutorials on techniques used in recent research. The tutorials here will take quite a while to get through—they are very comprehensive—but are well worth the effort to learn.

There are also a large number of algorithms that have been implemented for compatability with scikit-learn. These algorithms are not always included in scikit-learn itself for a number of reasons, but a list of many of these is maintained at `https://github.com/scikit-learn/scikit-learn/wiki/Third-party-projects-and-code-snippets`.

Extending the Jupyter Notebook

URL: `http://ipython.org/ipython-doc/1/interactive/public_server.html`

The Jupyter Notebook is a powerful tool. It can be extended in many ways, and one of those is to create a server to run your Notebooks, separately from your main computer. This is very useful if you use a low-power main computer, such as a small laptop, but have more powerful computers at your disposal. In addition, you can set up nodes to perform parallelized computations.

More datasets

URL: `http://archive.ics.uci.edu/ml/`

There are many datasets available on the Internet from a number of different sources. These include academic, commercial, and government datasets. A collection of well-labelled datasets is available at the UCI ML library, which is one of the best options to find datasets for testing your algorithms. Try out the OneR algorithm with some of these different datasets.

Other Evaluation Metrics

There is a wide range of evaluation metrics for other takes. Some notable ones to investigate are:

- The Lift Metric: `https://en.wikipedia.org/wiki/Lift_(data_mining)`
- Segment evaluation metrics: `http://segeval.readthedocs.io/en/latest/`
- Pearson's Correlation Coefficient: `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`
- Area under the ROC Curve: `http://gim.unmc.edu/dxtests/roc3.htm`
- Normalized Mutual Information: `http://scikit-learn.org/stable/modules/clustering.html#mutual-info-score`

Each of these metrics was developed with a particular application in mind. For example, the segment evaluation metrics evaluate how accurate breaking a document of text into chunks is, allowing for some variation between chunk boundaries. A good understanding of where evaluation metrics can be applied and where they can not is critical to ongoing success in data mining.

More application ideas

URL: `https://datapipeline.com.au/`

If you are looking for more ideas on data mining applications, specifically those for businesses, check out my company's blog. I post regularly about applications of data mining, focusing on practical outcomes for businesses.

Classifying with scikit-learn Estimators

A naïve implementation of the nearest neighbor algorithm is quite slow—it checks all pairs of points to find those that are close together. Better implementations exist, with some implemented in scikit-learn.

Scalability with the nearest neighbor

URL: `https://github.com/jnothman/scikit-learn/tree/pr2532`

For instance, a kd-tree can be created that speeds up the algorithm (and this is already included in scikit-learn).

Another way to speed up this search is to use locality-sensitive hashing, Locality-Sensitive Hashing (LSH). This is a proposed improvement for scikit-learn, and hasn't made it into the package at the time of writing. The preceding link gives a development branch of scikit-learn that will allow you to test out LSH on a dataset. Read through the documentation attached to this branch for details on doing this.

To install it, clone the repository and follow the instructions to install the Bleeding Edge code available at `http://scikit-learn.org/stable/install.html` on your computer. Remember to use the repository's code rather than the official source. I recommend that you use Anaconda for playing around with bleeding-edge packages so that they don't interfere with other libraries on your system.

More complex pipelines

URL: `http://scikit-learn.org/stable/modules/pipeline.html#featureunion-composite-feature-spaces`

The Pipelines we have used here follow a single stream—the output of one step is the input of another step.

Pipelines follow the transformer and estimator interfaces as well—this allows us to embed Pipelines within Pipelines. This is a useful construct for very complex models, but becomes very powerful when combined with Feature Unions, as shown in the preceding link.This allows us to extract multiple types of features at a time and then combine them to form a single dataset. For more details, see this example: `http://scikit-learn.org/stable/auto_examples/feature_stacker.html`.

Comparing classifiers

There are lots of classifiers in scikit-learn that are ready to use. The one you choose for a particular task is going to be based on a variety of factors. You can compare the f1-score to see which method is better, and you can investigate the deviation of those scores to see if that result is statistically significant.

An important factor is that they are trained and tested on the same data—that is, the test set for one classifier is the test set for all classifiers. Our use of random states allows us to ensure that this is the case—an important factor for replicating experiments.

Automated Learning

URL: `http://rhiever.github.io/tpot/`

URL: `https://github.com/automl/auto-sklearn`

It's almost cheating, but these packages will investigate a wide range of possible models for your data mining experiments for you. This removes the need to create a workflow testing a large number of parameters for a larger number of classifier types, and lets you focus on other things, such as feature extract--still critically important and not yet automated!

The general idea is that you extract your features and pass the resulting matrix onto one of these automated classification algorithms (or regression algorithms). It does the search for you and even exports the best model for you. In the case of TPOT, it even gives you Python code to create the model from scratch without having to install TPOT on your server.

Predicting Sports Winners with Decision Trees

URL: `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`

The pandas library is a great package—anything you normally write to do data loading is probably already implemented in pandas. You can learn more about it from their tutorial.

There is also a great blog post written by Chris Moffitt that overviews common tasks people do in Excel and how to do them in pandas: `http://pbpython.com/excel-pandas-comp.html`

You can also handle large datasets with pandas; see the answer, from user Jeff, to this StackOverflow question for an extensive overview of the process: `http://stackoverflow.com/a/14268804/307363`.

Another great tutorial on pandas is written by Brian Connelly: `http://bconnelly.net/2013/10/summarizing-data-in-python-with-pandas/`.

More complex features

URL: `http://www.basketball-reference.com/teams/ORL/2014_roster_status.html`

Larger exercise!

Sports teams change regularly from game to game. An easy win for a team can turn into a difficult game if a couple of the best players are suddenly injured. You can get the team rosters from basketball-reference as well. For example, the roster for the 2013-2014 season for the Orlando Magic is available at the preceding link. Similar data is available for all NBA teams.

Writing code to integrate how much a team changes and using that to add new features can improve the model significantly. This task will take quite a bit of work though!

Dask

URL: `http://dask.pydata.org/en/latest/`

If you want to take the features of pandas and increase its scalability, then Dask is for you. Dask provides parallelized versions of NumPy arrays, Pandas DataFrames, and task scheduling. Often, the interface is *nearly* the same as the original NumPy or Pandas versions.

Research

URL: `https://scholar.google.com.au/`

Larger exercise!As you might imagine, there has been a lot of work performed on predicting NBA games, as well as for all sports. Search "<SPORT> prediction" in Google Scholar to find research on predicting your favorite <SPORT>.

Recommending Movies Using Affinity Analysis

There are many recommendation-based datasets that are worth investigating, each with its own issues.

New datasets

URL: `http://www2.informatik.uni-freiburg.de/~cziegler/BX/`

Larger exercise!

There are many recommendation-based datasets that are worth investigating, each with its own issues. For example, the Book-Crossing dataset contains more than 278,000 users and over a million ratings. Some of these ratings are explicit (the user did give a rating), while others are more implicit. The weighting to these implicit ratings probably shouldn't be as high as for explicit ratings. The music website www.last.fm has released a great dataset for music recommendation: `http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/`.

There is also a joke recommendation dataset! See here: `http://eigentaste.berkeley.edu/dataset/`.

The Eclat algorithm

URL: `http://www.borgelt.net/eclat.html`

The APriori algorithm implemented here is easily the most famous of the association rule mining graphs, but isn't necessarily the best. Eclat is a more modern algorithm that can be implemented relatively easily.

Collaborative Filtering

URL: `https://github.com/python-recsys`

For those wanting to got much further with recommendation engines, it is necessary to investigate other formats for recommendations, such as collaborative filtering. This library provides some background into the algorithms and implementations, along with some tutorials. There is also a good overview at `http://blogs.gartner.com/martin-kihn/how-to-build-a-recommender-system-in-python/`.

Extracting Features with Transformers

Following topics, according to me, are also relevant when it comes to deeper understanding of Extracting Features with Transformers

Adding noise

We covered removing noise to improve features; however, improved performance can be obtained for some datasets by adding noise. The reason for this is simple—it helps stop overfitting by forcing the classifier to generalize its rules a little (although too much noise will make the model too general). Try implementing a Transformer that can add a given amount of noise to a dataset. Test that out on some of the datasets from UCI ML and see if it improves test-set performance.

Vowpal Wabbit

URL: `http://hunch.net/~vw/`

Vowpal Wabbit is a great project, providing very fast feature extraction for text-based problems. It comes with a Python wrapper, allowing you to call it from with Python code. Test it out on large datasets.

word2vec

URL: `https://radimrehurek.com/gensim/models/word2vec.html`

Word embeddings are receiving a lot of interest from research and industry, for a good reason: they perform very well on many text mining tasks. They are a big more complicated than the bag-of-words model and create larger models. Word embeddings are great features when you have lots of data and can even help in some cases with smaller amounts.

Social Media Insight Using Naive Bayes

Do consider the following points after finishing with Social Media Insight Using Native Bayes.

Spam detection

URL: `http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter`

Using the concepts here, you can create a spam detection method that is able to view a social media post and determine whether it is spam or not. Try this out by first creating a dataset of spam/not-spam posts, implementing the text mining algorithms, and then evaluating them.

One important consideration with spam detection is the false-positive/false-negative ratio. Many people would prefer to have a couple of spam messages slip through, rather than miss out on a legitimate message because the filter was too aggressive in stopping the spam. In order to turn your method for this, you can use a Grid Search with the f1-score as the evaluation criteria. See the preceding link for information on how to do this.

Natural language processing and part-of-speech tagging

URL: `http://www.nltk.org/book/ch05.html`

The techniques we used here are quite lightweight compared to some of the linguistic models employed in other areas. For example, part-of-speech tagging can help disambiguate word forms, allowing for higher accuracy. It comes with NLTK.

Discovering Accounts to Follow Using Graph Mining

Do give the following a read when done with the chapter.

More complex algorithms

URL: `https://www.cs.cornell.edu/home/kleinber/link-pred.pdf`Larger exercise!

There has been extensive research on predicting links in graphs, including for social networks. For instance, David Liben-Nowell and Jon Kleinberg published a paper on this topic that would serve as a great place for more complex algorithms, linked previously.

NetworkX

URL: `https://networkx.github.io/`

If you are going to be using graphs and networks more, going in-depth into the NetworkX package is well worth your time—the visualization options are great and the algorithms are well implemented. Another library called SNAP is also available with Python bindings, at `http://snap.stanford.edu/snappy/index.html`.

Beating CAPTCHAs with Neural Networks

You may find the following topics interesting as well:

Better (worse?) CAPTCHAs

URL: `http://scikit-image.org/docs/dev/auto_examples/applications/plot_geometric.html`

Larger exercise!

The CAPTCHAs we beat in this example were not as complex as those normally used today. You can create more complex variants using a number of techniques as follows:

- Applying different transformations such as the ones in scikit-image (see the preceding link)
- Using different colors and colors that don't translate well to grayscale
- Adding lines or other shapes to the image: `http://scikit-image.org/docs/dev/api/skimage.draw.html`

Deeper networks

These techniques will probably fool our current implementation, so improvements will need to be made to make the method better. Try some of the deeper networks we used. Larger networks need more data, though, so you will probably need to generate more than the few thousand samples we did here in order to get good performance. Generating these datasets is a good candidate for parallelization—lots of small tasks that can be performed independently.

A good idea for increasing your dataset size, which applies to other datasets as well, is to create variants of existing images. Flip images upside down, crop them weirdly, add noise, blur the image, make some random pixels black and so on.

Reinforcement learning

URL: `http://pybrain.org/docs/tutorial/reinforcement-learning.html`

Reinforcement learning is gaining traction as the next big thing in data mining—although it has been around a long time! PyBrain has some reinforcement learning algorithms that are worth checking out with this dataset (and others!).

Authorship Attribution

When it comes to Authorship Attribution do give the following topics a read.

Increasing the sample size

The Enron application we used ended up using just a portion of the overall dataset. There is lots more data available in this dataset. Increasing the number of authors will likely lead to a drop in accuracy, but it is possible to boost the accuracy further than was achieved here, using similar methods. Using a Grid Search, try different values for n-grams and different parameters for support vector machines, in order to get better performance on a larger number of authors.

Blogs dataset

The dataset used, provides authorship-based classes (each blogger ID is a separate author). This dataset can be tested using this kind of method as well. In addition, there are the other classes of gender, age, industry, and star sign that can be tested—are authorship-based methods good for these classification tasks?

Local n-grams

URL: `https://github.com/robertlayton/authorship_tutorials/blob/master/LNGTutorial.ipynb`

Another form of classifier is local n-gram, which involves choosing the best features per-author, not globally for the entire dataset. I wrote a tutorial on using local n-grams for authorship attribution, available at the preceding link.

Clustering News Articles

It won't hurt to read a little on the following topics

Clustering Evaluation

The evaluation of clustering algorithms is a difficult problem—on the one hand, we can sort of tell what good clusters look like; on the other hand, if we really know that, we should label some instances and use a supervised classifier! Much has been written on this topic. One slideshow on the topic that is a good introduction to the challenges follows: `http://www.cs.kent.edu/~jin/DM08/ClusterValidation.pdf`.

In addition, a very comprehensive (although now a little dated) paper on this topic is here: `http://web.itu.edu.tr/sgunduz/courses/verimaden/paper/validity_survey.pdf`.

The scikit-learn package does implement a number of the metrics described in those links, with an overview here: `http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation`.

Using some of these, you can start evaluating which parameters need to be used for better clusterings. Using a Grid Search, we can find parameters that maximize a metric—just like in classification.

Temporal analysis

Larger exercise!

The code we developed here can be rerun over many months. By adding some tags to each cluster, you can track which topics stay active over time, getting a longitudinal viewpoint of what is being discussed in the world news. To compare the clusters, consider a metric such as the adjusted mutual information score, which was linked to the scikit-learn documentation earlier. See how the clusters change after one month, two months, six months, and a year.

Real-time clusterings

The k-means algorithm can be iteratively trained and updated over time, rather than discrete analyses at given time frames. Cluster movement can be tracked in a number of ways—for instance, you can track which words are popular in each cluster and how much the centroids move per day. Keep the API limits in mind—you probably only need to do one check every few hours to keep your algorithm up-to-date.

Classifying Objects in Images Using Deep Learning

The following topics are also important when deeper study into Classifying objects is considered.

Mahotas

URL: `http://luispedro.org/software/mahotas/`

Another package for image processing is Mahotas, including better and more complex image processing techniques that can help achieve better accuracy, although they may come at a high computational cost. However, many image processing tasks are good candidates for parallelization. More techniques on image classification can be found in the research literature, with this survey paper as a good start: `http://ijarcce.com/upload/january/22-A%20Survey%20on%20Image%20Classification.pdf`.

Other image datasets are available at `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`.

There are many datasets of images available from a number of academic and industry-based sources. The linked website lists a bunch of datasets and some of the best algorithms to use on them. Implementing some of the better algorithms will require significant amounts of custom code, but the payoff can be well worth the pain.

Magenta

URL: `https://github.com/tensorflow/magenta/tree/master/magenta/reviews`

This repository contains a few high-quality deep learning papers that are worth reading, along with in-depth reviews of the paper and their techniques. If you want to go deep into deep learning, check out these papers first before expanding outwards.

Working with Big Data

The following resources on Big Data would be helpful

Courses on Hadoop

Both Yahoo and Google have great tutorials on Hadoop, which go from beginner to quite advanced levels. They don't specifically address using Python, but learning the Hadoop concepts and then applying them in Pydoop or a similar library can yield great results.

Yahoo's tutorial: `https://developer.yahoo.com/hadoop/tutorial/`

Google's tutorial: `https://cloud.google.com/hadoop/what-is-hadoop`

Pydoop

URL: `http://crs4.github.io/pydoop/tutorial/index.html`

Pydoop is a python library to run Hadoop jobs. Pydoop also works with HDFS, the Hadoop File System, although you can get that functionality in mrjob as well. Pydoop will give you a bit more control over running some jobs.

Recommendation engine

Building a large recommendation engine is a good test of your Big data skills. A great blog post by Mark Litwintschik covers an engine using Apache Spark, a big data technology: `http://tech.marksblogg.com/recommendation-engine-spark-python.html`

W.I.L.L

URL: `https://github.com/ironman5366/W.I.L.L`

Very large project!

This open source personal assistant can be your next JARVIS from Iron Man. You can add to this project using data mining techniques to allow it to learn to do some tasks that you need to do regularly. This is not easy, but the potential productivity gains are worth it.

More resources

The following would serve as a really good resource for additional information:

Kaggle competitions

URL: `www.kaggle.com/`

Kaggle runs data mining competitions regularly, often with monetary prizes.
Testing your skills on Kaggle competitions is a fast and great way to learn to work with real-world data mining problems. The forums are nice and share environments—often, you will see code released for a top-10 entry during the competition!

Coursera

URL: `www.coursera.org`

Coursera contains many courses on data mining and data science. Many of the courses are specialized, such as big data and image processing. A great general one to start with is Andrew Ng's famous course: `https://www.coursera.org/learn/machine-learning/`.
It is a bit more advanced than this and would be a great next step for interested readers. For neural networks, check out this course:
`https://www.coursera.org/course/neuralnets`.
If you complete all of these, try out the course on probabilistic graphical models at `https://www.coursera.org/course/pgm`.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y