Richard M. Reese,
Jennifer L. Reese

Java for
Data Science

Examine the techniques and Java tools supporting the
growing field of data science

Packt>

Java for Data Science

Table of Contents

Java for Data Science

Java for Data Science

Credits

Authors Richard M. Reese Jennifer L. Reese	**Copy Editors** Vikrant Phadkay Safis Editing
Reviewers Walter Molina Shilpi Saxena	**Project Coordinator** Nidhi Joshi
Commissioning Editor Veena Pagare	**Proofreader** Safis Editing
Acquisition Editor Tushar Gupta	**Indexer** Aishwarya Gangawane
Content Development Editor Aishwarya Pandere	**Graphics** Disha Haria
Technical Editor Suwarna Patil	**Production Coordinator** Nilesh Mohite

About the Authors

Richard M. Reese has worked in both industry and academics. For 17 years, he worked in the telephone and aerospace industries, serving in several capacities, including research and development, software development, supervision, and training. He currently teaches at Tarleton State University, where he has the opportunity to apply his years of industry experience to enhance his teaching.

Richard has written several Java books and a C Pointer book. He uses a concise and easy-to-follow approach to topics at hand. His Java books have addressed EJB 3.1, updates to Java 7 and 8, certification, jMonkeyEngine, natural language processing, functional programming, and networks.

Richard would like to thank his wife, Karla, for her continued support, and to the staff of Packt Publishing for their work in making this a better book.

Jennifer L. Reese studied computer science at Tarleton State University. She also earned her M.Ed. from Tarleton in December 2016. She currently teaches computer science to high-school students. Her research interests include the integration of computer science concepts with other academic disciplines, increasing diversity in computer science courses, and the application of data science to the field of education.

She previously worked as a software engineer developing software for county- and district-level government offices throughout Texas. In her free time she enjoys reading, cooking, and traveling —especially to any destination with a beach. She is a musician and appreciates a variety of musical genres.

I would like to thank Dad for his inspiration and guidance, Mom for her patience and perspective, and Jace for his support and always believing in me.

About the Reviewers

Walter Molina is a UI and UX developer from Villa Mercedes, San Luis, Argentina. His skills include, but are not limited to, HTML5, CSS3, and JavaScript. He uses these technologies at a Jedi/ninja level (along with a plethora of JavaScript libraries) in his daily work as a frontend developer at Tachuso, a creative content agency. He holds a bachelor's degree in computer science and is a member of the School of Engineering at local National University, where he teaches programming skills to second- and third-year students. His LinkedIn profile is https://ar.linkedin.com/in/waltermolina.

Shilpi Saxena is an IT professional and also a technology evangelist. She is an engineer who has had exposure to various domains (IOT and cloud computing space, healthcare, telecom, hiring, and manufacturing). She has experience in all the aspects of conception and execution of enterprise solutions. She has been architecting, managing, and delivering solutions in the big data space for the last 3 years; she also handles a high-performance and geographically distributed team of elite engineers.

Shilpi has more than 14 years (3 years in the big data space) of experience in the development and execution of various facets of enterprise solutions both in the products and services dimensions of the software industry. An engineer by degree and profession, she has worn various hats, such as developer, technical leader, product owner, tech manager, and so on, and has seen all the flavors that the industry has to offer. She has architected and worked through some of the pioneers' production implementations in big data on Storm and Impala with autoscaling in AWS.

Shilpi has also authored Real-time Analytics with Storm and Cassandra (https://www.packtpub.com/big-data-and-business-intelligence/learning-real-time-analytics-storm-and-cassandra) and Real time Big Data Analytics (https://www.packtpub.com/big-data-and-business-intelligence/real-time-big-data-analytics) with *Packt Publishing.*

www.PacktPub.com

eBooks, discount offers, and more

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 `customercare@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

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

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

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us**: `customerreviews@packtpub.com`.

Preface

In this book, we examine Java-based approaches to the field of data science. Data science is a broad topic and includes such subtopics as data mining, statistical analysis, audio and video analysis, and text analysis. A number of Java APIs provide support for these topics. The ability to apply these specific techniques allows for the creation of new, innovative applications able to handle the vast amounts of data available for analysis.

This book takes an expansive yet cursory approach to various aspects of data science. A brief introduction to the field is presented in the first chapter. Subsequent chapters cover significant aspects of data science, such as data cleaning and the application of neural networks. The last chapter combines topics discussed throughout the book to create a comprehensive data science application.

What this book covers

Chapter 1 , *Getting Started with Data Science* , provides an introduction to the technologies covered by the book. A brief explanation of each technology is given, followed by a short overview and demonstration of the support Java provides.

Chapter 2 , *Data Acquisition*, demonstrates how to acquire data from a number of sources, including Twitter, Wikipedia, and YouTube. The first step of a data science application is to acquire data.

Chapter 3 , *Data Cleaning* , explains that once data has been acquired, it needs to be cleaned. This can involve such activities as removing stop words, validating the data, and data conversion.

Chapter 4 , *Data Visualization* , shows that while numerical processing is a critical step in many data science tasks, people often prefer visual depictions of the results of analysis. This chapter demonstrates various Java approaches to this task.

Chapter 5 , *Statistical Data Analysis Techniques* , reviews basic statistical techniques, including regression analysis, and demonstrates how various Java APIs provide statistical support. Statistical analysis is key to many data analysis tasks.

Chapter 6 , *Machine Learning,* covers several machine learning algorithms, including decision trees and support vector machines. The abundance of available data provides an opportunity to apply machine learning techniques.

Chapter 7 , *Neural Networks* , explains that neural networks can be applied to solve a variety of data science problems. In this chapter, we explain how they work and demonstrate the use of several different types of neural networks.

Chapter 8 , *Deep Learning* , shows that deep learning algorithms are often described as multilevel neural networks. Java provides significant support in this area, and we will illustrate the use of this approach.

Chapter 9 , *Text Analysis* , explains that *s* ignificant portions of available datasets exist in textual formats. The field of natural language processing has advanced considerably and is frequently used in data science applications. We demonstrate various Java APIs used to support this type of analysis.

Chapter 10 , *Visual and Audio Analysis,* tells us that data science is not restricted to text processing. Many social media sites use visual data extensively. This chapter illustrates the Java supports available for this type of analysis.

Chapter 11 , *Mathematical and Parallel Techniques for Data Analysis* , investigates the support provided for low-level math operations and how they can be supported in a multiple processor environment. Data analysis, at its heart, necessitates the ability to manipulate and analyze large

quantities of numeric data.

Chapter 12 , *Bringing It All Together* , examines how the integration of the various technologies introduced in this book can be used to create a data science application. This chapter begins with data acquisition and incorporates many of the techniques used in subsequent chapters to build a complete application.

What you need for this book

Many of the examples in the book use Java 8 features. There are a number of Java APIs demonstrated, each of which is introduced before it is applied. An IDE is not required but is desirable.

Who this book is for

This book is aimed at experienced Java programmers who are interested in gaining a better understanding of the field of data science and how Java supports the underlying techniques. No prior experience in the field is needed.

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 are shown as follows: "The `getResult` method returns a `SpeechResult` instance which holds the result of the processing." Database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `KevinVoiceDirectory` contains two voices: `kevin` and `kevin16`."

A block of code is set as follows:

```
Voice[] voices = voiceManager.getVoices();
for (Voice v : voices) {
    out.println(v);
}
```

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

```
Name: kevin16
Description: default 16-bit diphone voice
Organization: cmu
Age: YOUNGER_ADULT
Gender: MALE
```

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: "Select the **Images** category and then filter for **Labeled for reuse**."

Note

Warnings or important notes appear in a box like this.

Tip

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/Java-for-Data-Science. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Errata

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

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the `Errata` section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

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

Chapter 1. Getting Started with Data Science

Data science is not a single science as much as it is a collection of various scientific disciplines integrated for the purpose of analyzing data. These disciplines include various statistical and mathematical techniques, including:

- Computer science
- Data engineering
- Visualization
- Domain-specific knowledge and approaches

With the advent of cheaper storage technology, more and more data has been collected and stored permitting previously unfeasible processing and analysis of data. With this analysis came the need for various techniques to make sense of the data. These large sets of data, when used to analyze data and identify trends and patterns, become known as **big data**.

This in turn gave rise to cloud computing and concurrent techniques such as **map-reduce,** which distributed the analysis process across a large number of processors, taking advantage of the power of parallel processing.

The process of analyzing big data is not simple and evolves to the specialization of developers who were known as **data scientists**. Drawing upon a myriad of technologies and expertise, they are able to analyze data to solve problems that previously were either not envisioned or were too difficult to solve.

Early big data applications were typified by the emergence of search engines capable of more powerful and accurate searches than their predecessors. For example, **AltaVista** was an early popular search engine that was eventually superseded by Google. While big data applications were not limited to these search engine functionalities, these applications laid the groundwork for future work in big data.

The term, data science, has been used since 1974 and evolved over time to include statistical analysis of data. The concepts of data mining and data analytics have been associated with data science. Around 2008, the term data scientist appeared and was used to describe a person who performs data analysis. A more in-depth discussion of the history of data science can be found at http://www.forbes.com/sites/gilpress/2013/05/28/a-very-short-history-of-data-science/#3d9ea08369fd.

This book aims to take a broad look at data science using Java and will briefly touch on many topics. It is likely that the reader may find topics of interest and pursue these at greater depth independently. The purpose of this book, however, is simply to introduce the reader to the significant data science topics and to illustrate how they can be addressed using Java.

There are many algorithms used in data science. In this book, we do not attempt to explain how they work except at an introductory level. Rather, we are more interested in explaining how they

can be used to solve problems. Specifically, we are interested in knowing how they can be used with Java.

Problems solved using data science

The various data science techniques that we will illustrate have been used to solve a variety of problems. Many of these techniques are motivated to achieve some economic gain, but they have also been used to solve many pressing social and environmental problems. Problem domains where these techniques have been used include finance, optimizing business processes, understanding customer needs, performing DNA analysis, foiling terrorist plots, and finding relationships between transactions to detect fraud, among many other data-intensive problems.

Data mining is a popular application area for data science. In this activity, large quantities of data are processed and analyzed to glean information about the dataset, to provide meaningful insights, and to develop meaningful conclusions and predictions. It has been used to analyze customer behavior, detecting relationships between what may appear to be unrelated events, and to make predictions about future behavior.

Machine learning is an important aspect of data science. This technique allows the computer to solve various problems without needing to be explicitly programmed. It has been used in self-driving cars, speech recognition, and in web searches. In data mining, the data is extracted and processed. With machine learning, computers use the data to take some sort of action.

Understanding the data science problem - solving approach

Data science is concerned with the processing and analysis of large quantities of data to create models that can be used to make predictions or otherwise support a specific goal. This process often involves the building and training of models. The specific approach to solve a problem is dependent on the nature of the problem. However, in general, the following are the high-level tasks that are used in the analysis process:

- **Acquiring the data**: Before we can process the data, it must be acquired. The data is frequently stored in a variety of formats and will come from a wide range of data sources.
- **Cleaning the data**: Once the data has been acquired, it often needs to be converted to a different format before it can be used. In addition, the data needs to be processed, or cleaned, so as to remove errors, resolve inconsistencies, and otherwise put it in a form ready for analysis.
- **Analyzing the data**: This can be performed using a number of techniques including:
  - **Statistical analysis**: This uses a multitude of statistical approaches to provide insight into data. It includes simple techniques and more advanced techniques such as regression analysis.
  - **AI analysis**: These can be grouped as machine learning, neural networks, and deep learning techniques:
    - Machine learning approaches are characterized by programs that can learn without being specifically programmed to complete a specific task
    - Neural networks are built around models patterned after the neural connection of the brain
    - Deep learning attempts to identify higher levels of abstraction within a set of data

  - **Text analysis**: This is a common form of analysis, which works with natural languages to identify features such as the names of people and places, the relationship between parts of text, and the implied meaning of text.
  - **Data visualization**: This is an important analysis tool. By displaying the data in a visual form, a hard-to-understand set of numbers can be more readily understood.
  - **Video, image, and audio processing and analysis**: This is a more specialized form of analysis, which is becoming more common as better analysis techniques are discovered and faster processors become available. This is in contrast to the more common text processing and analysis tasks.

Complementing this set of tasks is the need to develop applications that are efficient. The introduction of machines with multiple processors and GPUs contributes significantly to the end result.

While the exact steps used will vary by application, understanding these basic steps provides the basis for constructing solutions to many data science problems.

Using Java to support data science

Java and its associated third-party libraries provide a range of support for the development of data science applications. There are numerous core Java capabilities that can be used, such as the basic string processing methods. The introduction of lambda expressions in Java 8 helps enable more powerful and expressive means of building applications. In many of the examples that follow in subsequent chapters, we will show alternative techniques using lambda expressions.

There is ample support provided for the basic data science tasks. These include multiple ways of acquiring data, libraries for cleaning data, and a wide variety of analysis approaches for tasks such as natural language processing and statistical analysis. There are also myriad of libraries supporting neural network types of analysis.

Java can be a very good choice for data science problems. The language provides both object-oriented and functional support for solving problems. There is a large developer community to draw upon and there exist multiple APIs that support data science tasks. These are but a few reasons as to why Java should be used.

The remainder of this chapter will provide an overview of the data science tasks and Java support demonstrated in the book. Each section is only able to present a brief introduction to the topics and the available support. The subsequent chapter will go into considerably more depth regarding these topics.

Acquiring data for an application

Data acquisition is an important step in the data analysis process. When data is acquired, it is often in a specialized form and its contents may be inconsistent or different from an application's need. There are many sources of data, which are found on the Internet. Several examples will be demonstrated in Chapter 2, *Data Acquisition*.

Data may be stored in a variety of formats. Popular formats for text data include HTML, **Comma Separated Values** (**CSV**), **JavaScript Object Notation** (**JSON**), and XML. Image and audio data are stored in a number of formats. However, it is frequently necessary to convert one data format into another format, typically plain text.

For example, JSON (http://www.JSON.org/) is stored using blocks of curly braces containing key-value pairs. In the following example, parts of a YouTube result is shown:

```
{
  "kind": "youtube#searchResult",
  "etag": etag,
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  ...
}
```

Data is acquired using techniques such as **processing live streams**, **downloading compressed files**, and through **screen scraping**, where the information on a web page is extracted. **Web crawling** is a technique where a program examines a series of web pages, moving from one page to another, acquiring the data that it needs.

With many popular media sites, it is necessary to acquire a user ID and password to access data. A commonly used technique is **OAuth,** which is an open standard used to authenticate users to many different websites. The technique delegates access to a server resource and works over HTTPS. Several companies use OAuth 2.0, including PayPal, Facebook, Twitter, and Yelp.

The importance and process of cleaning data

Once the data has been acquired, it will need to be cleaned. Frequently, the data will contain errors, duplicate entries, or be inconsistent. It often needs to be converted to a simpler data type such as text. **Data cleaning** is often referred to as **data wrangling**, **reshaping,** or **munging**. They are effectively synonyms.

When data is cleaned, there are several tasks that often need to be performed, including checking its validity, accuracy, completeness, consistency, and uniformity. For example, when the data is incomplete, it may be necessary to provide substitute values.

Consider CSV data. It can be handled in one of several ways. We can use simple Java techniques such as the `String` class' `split` method. In the following sequence, a string array, `csvArray`, is assumed to hold comma-delimited data. The `split` method populates a second array, `tokenArray`.

```java
for(int i=0; i<csvArray.length; i++) {
    tokenArray[i] = csvArray[i].split(",");
}
```

More complex data types require APIs to retrieve the data. For example, in Chapter 3, *Data Cleaning*, we will use the Jackson Project (https://github.com/FasterXML/jackson) to retrieve fields from a JSON file. The example uses a file containing a JSON-formatted presentation of a person, as shown next:

```json
{
 "firstname":"Smith",
 "lastname":"Peter",
 "phone":8475552222,
 "address":["100 Main Street","Corpus","Oklahoma"]
}
```

The code sequence that follows shows how to extract the values for fields of a person. A parser is created, which uses `getCurrentName` to retrieve a field name. If the name is `firstname`, then the `getText` method returns the value for that field. The other fields are handled in a similar manner.

```java
try {
    JsonFactory jsonfactory = new JsonFactory();
    JsonParser parser = jsonfactory.createParser(
        new File("Person.json"));
    while (parser.nextToken() != JsonToken.END_OBJECT) {
        String token = parser.getCurrentName();
        if ("firstname".equals(token)) {
            parser.nextToken();
            String fname = parser.getText();
            out.println("firstname : " + fname);
        }
        ...
```

```
    }
    parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

The output of this example is as follows:

firstname : Smith

Simple data cleaning may involve converting the text to lowercase, replacing certain text with blanks, and removing multiple whitespace characters with a single blank. One way of doing this is shown next, where a combination of the `String` class' `toLowerCase`, `replaceAll`, and `trim` methods are used. Here, a string containing dirty text is processed:

```
dirtyText = dirtyText
    .toLowerCase()
    .replaceAll("[\\d[^\\w\\s]]+", "
    .trim();
while(dirtyText.contains("  ")){
    dirtyText = dirtyText.replaceAll("  ", " ");
}
```

Stop words are words such as *the, and,* or *but* that do not always contribute to the analysis of text. Removing these stop words can often improve the results and speed up the processing.

The **LingPipe API** can be used to remove stop words. In the next code sequence, a `TokenizerFactory` class instance is used to tokenize text. Tokenization is the process of returning individual words. The `EnglishStopTokenizerFactory` class is a special tokenizer that removes common English stop words.

```
text = text.toLowerCase().trim();
TokenizerFactory fact = IndoEuropeanTokenizerFactory.INSTANCE;
fact = new EnglishStopTokenizerFactory(fact);
Tokenizer tok = fact.tokenizer(
    text.toCharArray(), 0, text.length());
for(String word : tok){
    out.print(word + " ");
}
```

Consider the following text, which was pulled from the book, Moby Dick:

```
Call me Ishmael. Some years ago- never mind how long precisely - having
little or no money in my purse, and nothing particular to interest me on
shore, I thought I would sail about a little and see the watery part of the
world.
```

The output will be as follows:

call me ishmael . years ago - never mind how long precisely - having little money my purse , nothing particular interest me shore , i thought i sail little see watery part world .

These are just a couple of the data cleaning tasks discussed in [Chapter 3](#), *Data Cleaning.*

Visualizing data to enhance understanding

The analysis of data often results in a series of numbers representing the results of the analysis. However, for most people, this way of expressing results is not always intuitive. A better way to understand the results is to create graphs and charts to depict the results and the relationship between the elements of the result.
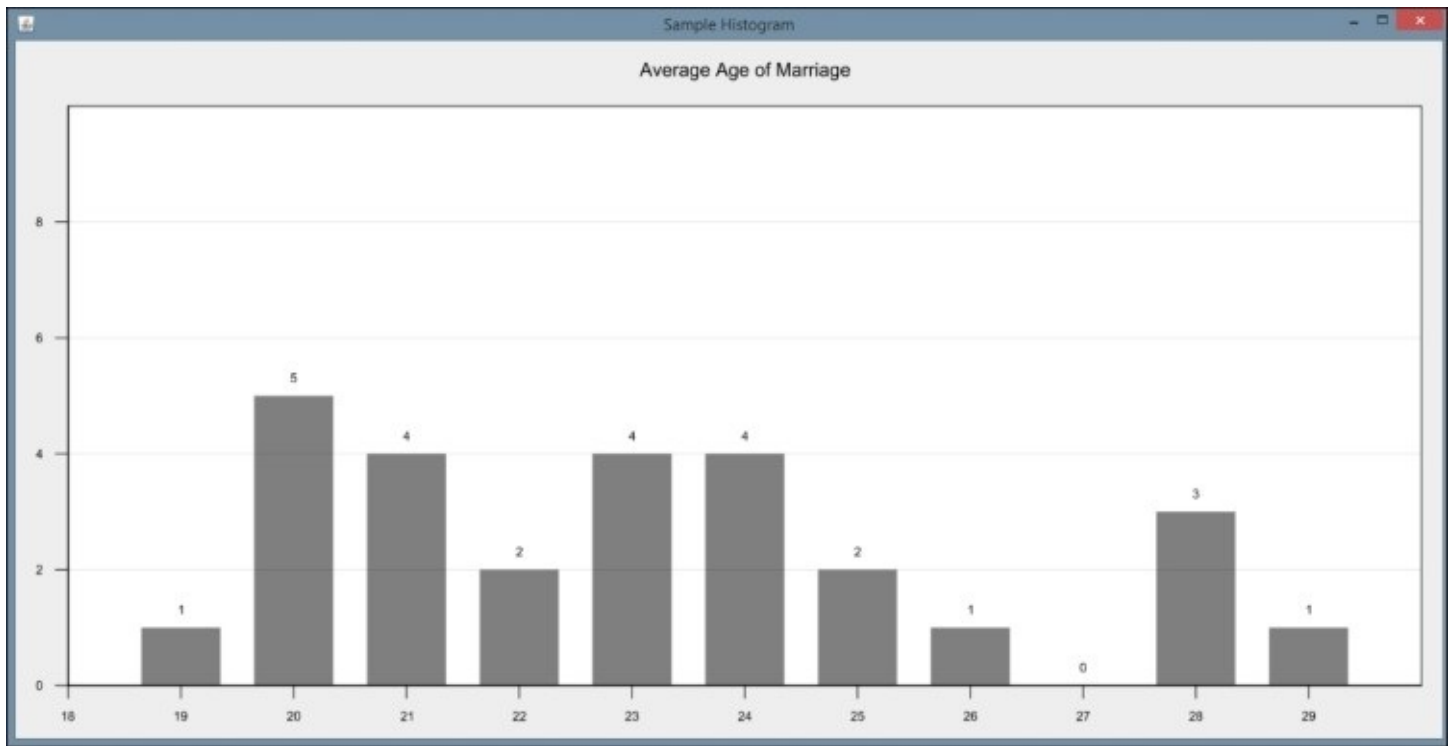
The human mind is often good at seeing patterns, trends, and outliers in visual representation. The large amount of data present in many data science problems can be analyzed using visualization techniques. Visualization is appropriate for a wide range of audiences ranging from analysts to upper-level management to clientele. In this chapter, we present various visualization techniques and demonstrate how they are supported in Java.

In Chapter 4, *Data Visualization,* we illustrate how to create different types of graphs, plots, and charts. These examples use JavaFX using a free library called **GRAL**(http://trac.erichseifert.de/gral/).

Visualization allows users to examine large datasets in ways that provide insights that are not present in the mass of the data. Visualization tools helps us identify potential problems or unexpected data results and develop meaningful interpretations of the data.

For example, outliers, which are values that lie outside of the normal range of values, can be hard to spot from a sea of numbers. Creating a graph based on the data allows users to quickly see outliers. It can also help spot errors quickly and more easily classify data.

For example, the following chart might suggest that the upper two values should be outliers that need to be dealt with:
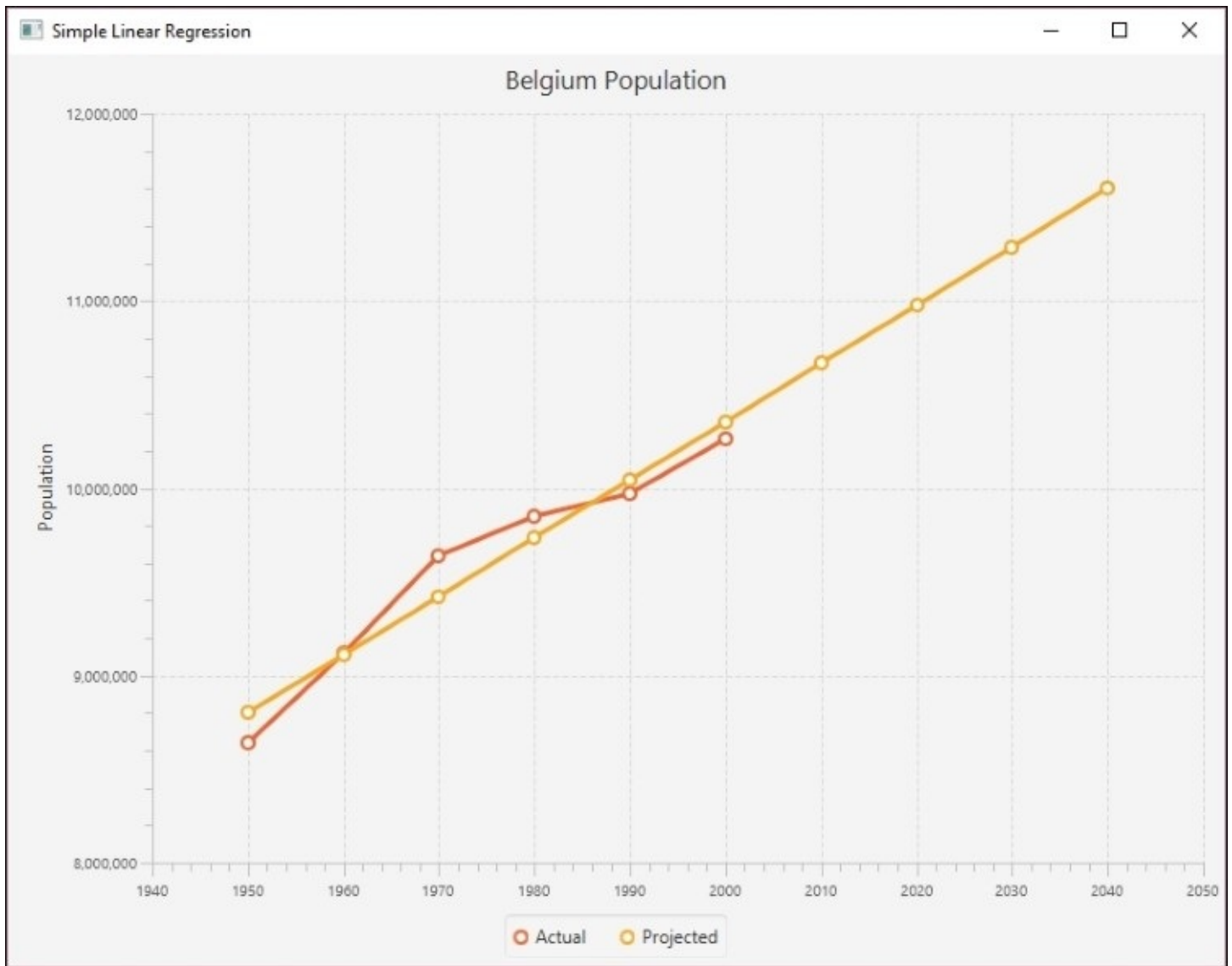
The use of statistical methods in data science

Statistical analysis is the key to many data science tasks. It is used for many types of analysis ranging from the computation of simple mean and medium to complex multiple regression analysis. Chapter 5, *Statistical Data Analysis Techniques,* introduces this type of analysis and the Java support available.

Statistical analysis is not always an easy task. In addition, advanced statistical techniques often require a particular mindset to fully comprehend, which can be difficult to learn. Fortunately, many techniques are not that difficult to use and various libraries mitigate some of these techniques' inherent complexity.

Regression analysis, in particular, is an important technique for analyzing data. The technique attempts to draw a line that matches a set of data. An equation representing the line is calculated and can be used to predict future behavior. There are several types of regression analysis, including simple and multiple regression. They vary by the number of variables being considered.

The following graph shows the straight line that closely matches a set of data points representing the population of Belgium over several decades:

Simple statistical techniques, such as mean and standard deviation, can be computed using basic Java. They can also be handled by libraries such as Apache Commons. For example, to calculate the median, we can use the Apache Commons `DescriptiveStatistics` class. This is illustrated next where the median of an array of doubles is calculated. The numbers are added to an instance of this class, as shown here:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2,
    12.5, 17.8, 16.5, 12.5};
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
```

The `getPercentile` method returns the value stored at the percentile specified in its argument. To find the median, we use the value of `50`.

```
out.println("The median is " + statTest.getPercentile(50));
```

Our output is as follows:

The median is 16.2

In Chapter 5, *Statistical Data Analysis Techniques*, we will demonstrate how to perform regression analysis using the Apache Commons `SimpleRegression` class.

Machine learning applied to data science

Machine learning has become increasingly important for data science analysis as it has been for a multitude of other fields. A defining characteristic of machine learning is the ability of a model to be trained on a set of representative data and then later used to solve similar problems. There is no need to explicitly program an application to solve the problem. A model is a representation of the real-world object.

For example, customer purchases can be used to train a model. Subsequently, predictions can be made about the types of purchases a customer might subsequently make. This allows an organization to tailor ads and coupons for a customer and potentially providing a better customer experience.

Training can be performed in one of several different approaches:

- **Supervised learning**: The model is trained with annotated, labeled, data showing corresponding correct results
- **Unsupervised learning**: The data does not contain results, but the model is expected to find relationships on its own
- **Semi-supervised**: A small amount of labeled data is combined with a larger amount of unlabeled data
- **Reinforcement learning**: This is similar to supervised learning, but a reward is provided for good results

There are several approaches that support machine learning. In Chapter 6, *Machine Learning,* we will illustrate three techniques:

- **Decision trees**: A tree is constructed using features of the problem as internal nodes and the results as leaves
- **Support vector machines**: This is used for classification by creating a hyperplane that partitions the dataset and then makes predictions
- **Bayesian networks**: This is used to depict probabilistic relationships between events

A **Support Vector Machine** (**SVM**) is used primarily for classification type problems. The approach creates a hyperplane to categorize data, which can be envisioned as a **geometric plane** that separates two regions. In a two-dimensional space, it will be a line. In a three-dimensional space, it will be a two-dimensional plane. In Chapter 6, *Machine Learning,* we will demonstrate how to use the approach using a set of data relating to the propensity of individuals to camp. We will use the Weka class, `SMO`, to demonstrate this type of analysis.

The following figure depicts a hyperplane using a distribution of two types of data points. The lines represent possible hyperplanes that separate these points. The lines clearly separate the data points except for one outlier.
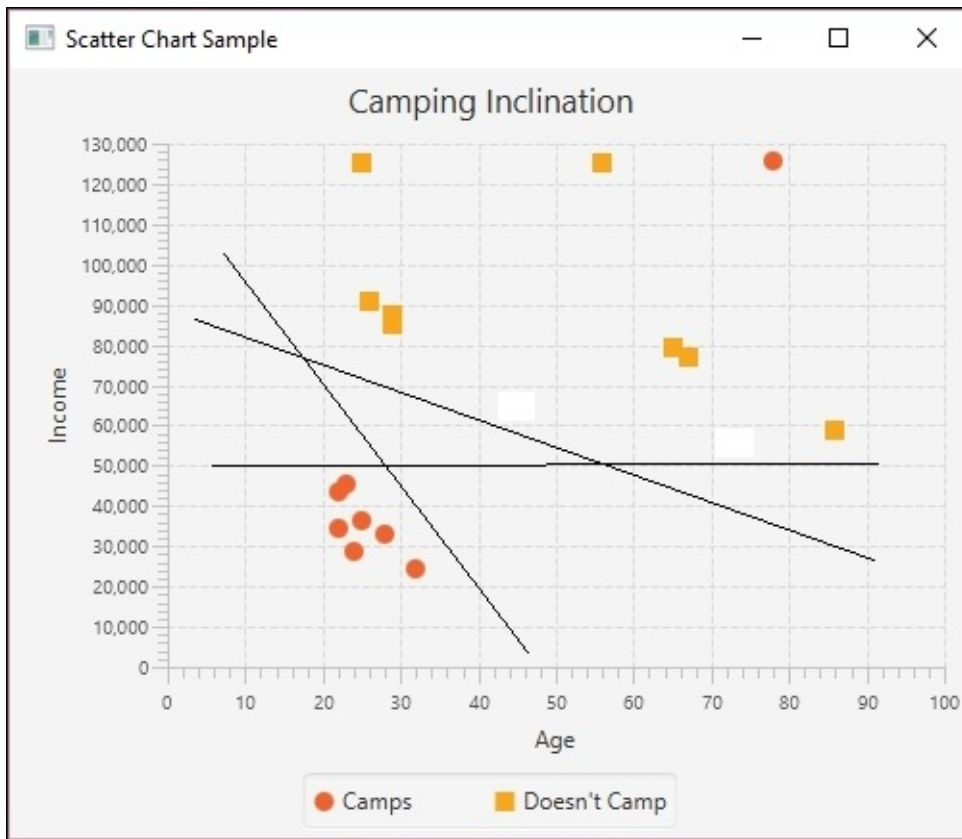
Once the model has been trained, the possible hyperplanes are considered and predictions can then be made using similar data.

Using neural networks in data science

An **Artificial Neural Network** (**ANN**), which we will call a **neural network**, is based on the neuron found in the brain. A **neuron** is a cell that has **dendrites** connecting it to input sources and other neurons. Depending on the input source, a weight allocated to a source, the neuron is activated, and then **fires** a signal down a dendrite to another neuron. A collection of neurons can be trained to respond to a set of input signals.

An artificial neuron is a node that has one or more inputs and a single output. Each input has a **weight** assigned to it that can change over time. A neural network can learn by feeding an input into a network, invoking an **activation function**, and comparing the results. This function combines the inputs and creates an output. If outputs of multiple neurons match the expected result, then the network has been trained correctly. If they don't match, then the network is modified.

A neural network can be visualized as shown in the following figure, where **Hidden Layer** is used to augment the process:
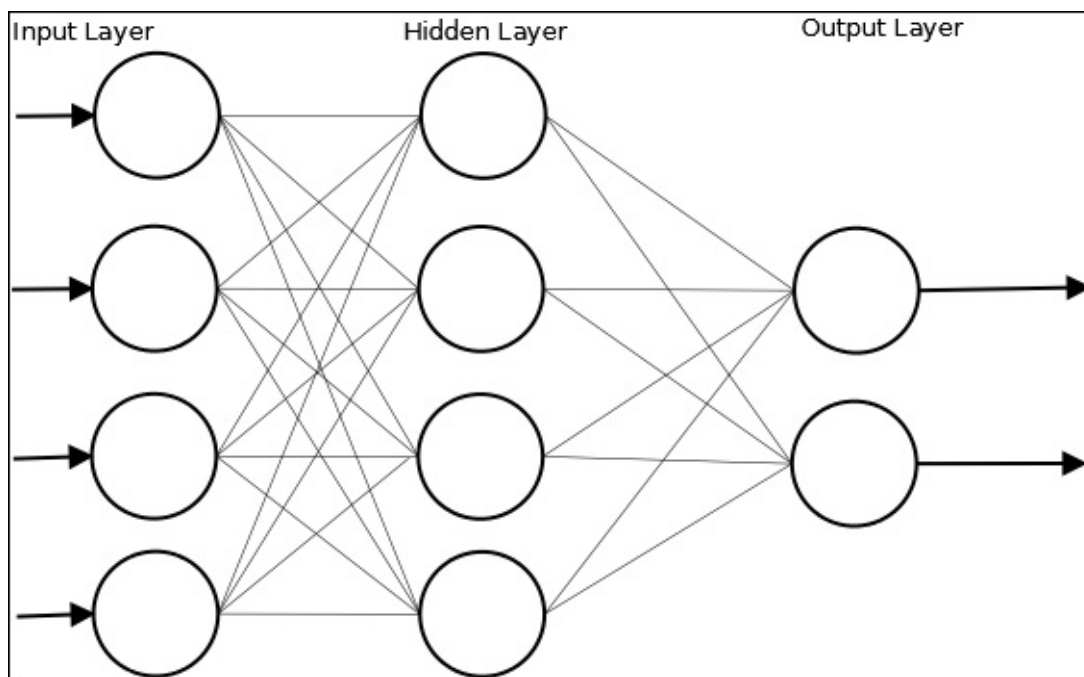
In [Chapter 7](), *Neural Networks*, we will use the `Weka` class, `MultilayerPerceptron`, to illustrate the creation and use of a **Multi Layer Perceptron** (**MLP**) network. As we will explain, this type of network is a feedforward neural network with multiple layers. The network uses supervised learning with backpropagation. The example uses a dataset called `dermatology.arff` that contains *366* instances that are used to diagnose erythemato-squamous diseases. It uses *34* attributes to classify the disease into one of the five different categories.

The dataset is split into a training set and a testing set. Once the data has been read, the MLP

instance is created and initialized using the method to configure the attributes of the model, including how quickly the model is to learn and the amount of time spent training the model.

```
String trainingFileName = "dermatologyTrainingSet.arff";
String testingFileName = "dermatologyTestingSet.arff";

try (FileReader trainingReader = new FileReader(trainingFileName);
        FileReader testingReader =
            new FileReader(testingFileName)) {
    Instances trainingInstances = new Instances(trainingReader);
    trainingInstances.setClassIndex(
        trainingInstances.numAttributes() - 1);
    Instances testingInstances = new Instances(testingReader);
    testingInstances.setClassIndex(
        testingInstances.numAttributes() - 1);

    MultilayerPerceptron mlp = new MultilayerPerceptron();
    mlp.setLearningRate(0.1);
    mlp.setMomentum(0.2);
    mlp.setTrainingTime(2000);
    mlp.setHiddenLayers("3");
        mlp.buildClassifier(trainingInstances);
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

The model is then evaluated using the testing data:

```
Evaluation evaluation = new Evaluation(trainingInstances);
evaluation.evaluateModel(mlp, testingInstances);
```

The results can then be displayed:

```
System.out.println(evaluation.toSummaryString());
```

The truncated output of this example is shown here where the number of correctly and incorrectly identified diseases are listed:

Correctly Classified Instances 73 98.6486 %

Incorrectly Classified Instances 1 1.3514 %

The various attributes of the model can be tweaked to improve the model. In Chapter 7, *Neural Networks*, we will discuss this and other techniques in more depth.

Deep learning approaches

Deep learning networks are often described as neural networks that use multiple intermediate layers. Each layer will train on the outputs of a previous layer potentially identifying features and subfeatures of a dataset. The features refer to those aspects of the data that may be of interest. In [Chapter 8](#), *Deep Learning,* we will examine these types of networks and how they can support several different data science tasks.

These networks often work with unstructured and unlabeled datasets, which is the vast majority of the data available today. A typical approach is to take the data, identify features, and then use these features and their corresponding layers to reconstruct the original dataset, thus validating the network. The **Restricted Boltzmann Machines** (**RBM**) is a good example of the application of this approach.

The deep learning network needs to ensure that the results are accurate and minimizes any error that can creep into the process. This is accomplished by adjusting the internal weights assigned to neurons based on what is known as **gradient descent**. This represents the slope of the weight changes. The approach modifies the weight so as to minimize the error and also speeds up the learning process.

There are several types of networks that have been classified as a deep learning network. One of these is an **autoencoder** network. In this network, the layers are symmetrical where the number of input values is the same as the number of output values and the intermediate layers effectively compress the data to a single smaller internal layer. Each layer of the autoencoder is a RBM.

This structure is reflected in the following example, which will extract the numbers found in a set of images containing hand-written numbers. The details of the complete example are not shown here, but notice that 1,000 input and output values are used along with internal layers consisting of RBMs. The size of the layers are specified in the `nOut` and `nIn` methods.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(numberOfIterations)
        .optimizationAlgo(
          OptimizationAlgorithm.LINE_GRADIENT_DESCENT)
        .list()
        .layer(0, new RBM.Builder()
           .nIn(numberOfRows * numberOfColumns).nOut(1000)
           .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
           .build())
        .layer(1, new RBM.Builder().nIn(1000).nOut(500)
           .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
           .build())
        .layer(2, new RBM.Builder().nIn(500).nOut(250)
           .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
           .build())
        .layer(3, new RBM.Builder().nIn(250).nOut(100)
           .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
```

```
        .build())
    .layer(4, new RBM.Builder().nIn(100).nOut(30)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build()) //encoding stops
    .layer(5, new RBM.Builder().nIn(30).nOut(100)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build()) //decoding starts
    .layer(6, new RBM.Builder().nIn(100).nOut(250)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(7, new RBM.Builder().nIn(250).nOut(500)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(8, new RBM.Builder().nIn(500).nOut(1000)
        .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
        .build())
    .layer(9, new OutputLayer.Builder(
            LossFunctions.LossFunction.RMSE_XENT).nIn(1000)
            .nOut(numberOfRows * numberOfColumns).build())
    .pretrain(true).backprop(true)
    .build();
```

Once the model has been trained, it can be used for predictive and searching tasks. With a search, the compressed middle layer can be used to match other compressed images that need to be classified.

Performing text analysis

The field of **Natural Language Processing** (**NLP**) is used for many different tasks including text searching, language translation, sentiment analysis, speech recognition, and classification to mention a few. Processing text is difficult due to a number of reasons, including the inherent ambiguity of natural languages.

There are several different types of processing that can be performed such as:

- **Identifying Stop words**: These are words that are common and may not be necessary for processing
- **Name Entity Recognition** (**NER**): This is the process of identifying elements of text such as people's names, location, or things
- **Parts of Speech** (**POS**): This identifies the grammatical parts of a sentence such as noun, verb, adjective, and so on
- **Relationships**: Here we are concerned with identifying how parts of text are related to each other, such as the subject and object of a sentence

As with most data science problems, it is important to preprocess and clean text. In Chapter 9, *Text Analysis*, we examine the support Java provides for this area of data science.

For example, we will use Apache's OpenNLP (https://opennlp.apache.org/) library to find the parts of speech. This is just one of the several NLP APIs that we could have used including LingPipe (http://alias-i.com/lingpipe/), Apache UIMA (https://uima.apache.org/), and Standford NLP (http://nlp.stanford.edu/). We chose OpenNLP because it is easy to use for this example.

In the following example, a model used to identify POS elements is found in the `en-pos-maxent.bin` file. An array of words is initialized and the POS model is created:

```
try (InputStream input = new FileInputStream(
        new File("en-pos-maxent.bin"));) {
    String sentence = "Let's parse this sentence.";
    ...
    String[] words;
    ...
    list.toArray(words);
    POSModel posModel = new POSModel(input);
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The `tag` method is passed an array of `words` and returns an array of tags. The words and tags are then displayed.

```
String[] posTags = posTagger.tag(words);
for(int i=0; i<posTags.length; i++) {
    out.println(words[i] + " - " + posTags[i]);
}
```

The output for this example is as follows:

```
Let's - NNP

parse - NN

this - DT

sentence. - NN
```

The abbreviations `NNP` and `DT` stand for a singular proper noun and determiner respectively. We examine several other NLP techniques in Chapter 9, *Text Analysis*.

Visual and audio analysis

In [Chapter 10](#), *Visual and Audio Analysis,* we demonstrate several Java techniques for processing sounds and images. We begin by demonstrating techniques for sound processing, including speech recognition and text-to-speech APIs. Specifically, we will use the FreeTTS (http://freetts.sourceforge.net/docs/index.php) API to convert text to speech. We also include a demonstration of the **CMU Sphinx** toolkit for speech recognition.

The **Java Speech API** (**JSAPI**) (http://www.oracle.com/technetwork/java/index-140170.html) supports speech technology. This API, created by third-party vendors, supports speech recognition and speech synthesizers. FreeTTS and Festival (http://www.cstr.ed.ac.uk/projects/festival/) are examples of vendors supporting JSAPI.

In the second part of the chapter, we examine image processing techniques such as facial recognition. This demonstration involves identifying faces within an image and is easy to accomplish using OpenCV (http://opencv.org/).

Also, in [Chapter 10](#), *Visual and Audio Analysis,* we demonstrate how to extract text from images, a process known as **OCR**. A common data science problem involves extracting and analyzing text embedded in an image. For example, the information contained in license plate, road signs, and directions can be significant.

In the following example, explained in more detail in [Chapter 11](#), *Mathematical and Parallel Techniques for Data Analysis* accomplishes OCR using Tess4j (http://tess4j.sourceforge.net/) a Java JNA wrapper for Tesseract OCR API. We perform OCR on an image captured from the Wikipedia article on OCR (https://en.wikipedia.org/wiki/Optical_character_recognition#Applications), shown here:

The `ITesseract` interface provides numerous OCR methods. The `doOCR` method takes a file and returns a string containing the words found in the file as shown here:

```
ITesseract instance = new Tesseract();
try {
```

```
    String result = instance.doOCR(new File("OCRExample.png"));
    System.out.println(result);
} catch (TesseractException e) {
    System.err.println(e.getMessage());
}
```

A part of the output is shown next:

OCR engines nave been developed into many lunds oiobiectorlented OCR applicatlons, sucn as reoeipt OCR, involoe OCR, check OCR, legal billing document OCR

They can be used ior

- Data entry ior business documents, e g check, passport, involoe, bank statement and receipt

- Automatic number plate recognnlon

As you can see, there are numerous errors in this example that need to be addressed. We build upon this example in Chapter 11, *Mathematical and Parallel Techniques for Data Analysis*, with a discussion of enhancements and considerations to ensure the OCR process is as effective as possible.

We will conclude the chapter with a discussion of NeurophStudio, a neural network Java-based editor, to classify images and perform image recognition. We train a neural network to recognize and classify faces in this section.

Improving application performance using parallel techniques

In [Chapter 11](#), *Mathematical and Parallel Techniques for Data Analysis,* we consider some of the parallel techniques available for data science applications. Concurrent execution of a program can significantly improve performance. In relation to data science, these techniques range from low-level mathematical calculations to higher-level API-specific options.

This chapter includes a discussion of basic performance enhancement considerations. Algorithms and application architecture matter as much as enhanced code, and this should be considered when attempting to integrate parallel techniques. If an application does not behave in the expected or desired manner, any gains from parallel optimizing are irrelevant.

Matrix operations are essential to many data applications and supporting APIs. We will include a discussion in this chapter about matrix multiplication and how it is handled using a variety of approaches. Even though these operations are often hidden within the API, it can be useful to understand how they are supported.

One approach we demonstrate utilizes the Apache Commons Math API (http://commons.apache.org/proper/commons-math/). This API supports a large number of mathematical and statistical operations, including matrix multiplication. The following example illustrates how to perform matrix multiplication.

We first declare and initialize matrices `A` and `B`:

```
double[][] A = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};

double[][] B = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
```

Apache Commons uses the `RealMatrix` class to store a matrix. Next, we use the `Array2DRowRealMatrix` constructor to create the corresponding matrices for `A` and `B`:

```
RealMatrix aRealMatrix = new Array2DRowRealMatrix(A);
RealMatrix bRealMatrix = new Array2DRowRealMatrix(B);
```

We perform multiplication simply using the `multiply` method:

```
RealMatrix cRealMatrix = aRealMatrix.multiply(bRealMatrix);
```

Finally, we use a `for` loop to display the results:

```
for (int i = 0; i < cRealMatrix.getRowDimension(); i++) {
    System.out.println(cRealMatrix.getRowVector(i));
}
```

The output is as follows:

```
{0.02761552; 0.19992684; 0.2131916}
{0.14428772; 0.41179806; 0.54165016}
{0.34857924; 0.32834382; 0.70581912}
{0.19639854; 0.29411363; 0.4963528}
```

Another approach to concurrent processing involves the use of Java threads. Threads are used by APIs such as Aparapi when multiple CPUs or GPUs are not available.

Data science applications often take advantage of the map-reduce algorithm. We will demonstrate parallel processing by using Apache's Hadoop to perform map-reduce. Designed specifically for large datasets, Hadoop reduces processing time for large scale data science projects. We demonstrate a technique for calculating the average value of a large dataset.

We also include examples of APIs that support multiple processors, including CUDA and OpenCL. CUDA is supported using Java bindings for CUDA (JCuda) (http://jcuda.org/). We also discuss OpenCL and its Java support. The Aparapi API provides high-level support for using multiple CPUs or GPUs and we include a demonstration of Aparapi in support of matrix multiplication.

Assembling the pieces

In the final chapter of this book, we will tie together many of the techniques explored in the previous chapters. We will create a simple console-based application for acquiring data from Twitter and performing various types of data manipulation and analysis. Our goal in this chapter is to demonstrate a simple project exploring a variety of data science concepts and provide insights and considerations for future projects.

Specifically, the application developed in the final chapter performs several high-level tasks, including data acquisition, data cleaning, sentiment analysis, and basic statistical collection. We demonstrate these techniques using Java 8 Streams and focus on Java 8 approaches whenever possible.

Summary

Data science is a broad, diverse field of study and it would be impossible to explore exhaustively within this book. We hope to provide a solid understanding of important data science concepts and equip the reader for further study. In particular, this book will provide concrete examples of different techniques for all stages of data science related inquiries. This ranges from data acquisition and cleaning to detailed statistical analysis.

So let's start with a discussion of data acquisition and how Java supports it as illustrated in the next chapter.

Chapter 2. Data Acquisition

It is never much fun to work with code that is not formatted properly or uses variable names that do not convey their intended purpose. The same can be said of data, except that bad data can result in inaccurate results. Thus, data acquisition is an important step in the analysis of data. Data is available from a number of sources but must be retrieved and ultimately processed before it can be useful. It is available from a variety of sources. We can find it in numerous public data sources as simple files, or it may be found in more complex forms across the Internet. In this chapter, we will demonstrate how to acquire data from several of these, including various Internet sites and several social media sites.

We can access data from the Internet by downloading specific files or through a process known as **web scraping**, which involves extracting the contents of a web page. We also explore a related topic known as **web crawling**, which involves applications that examine a web site to determine whether it is of interest and then follows embedded links to identify other potentially relevant pages.

We can also extract data from social media sites. These types of sites often hold a treasure trove of data that is readily available if we know how to access it. In this chapter, we will demonstrate how to extract data from several sites, including:

- Twitter
- Wikipedia
- Flickr
- YouTube

When extracting data from a site, many different data formats may be encountered. We will examine three basic types: text, audio, and video. However, even within text, audio, and video data, many formats exist. For audio data alone, there are 45 audio coding formats compared at https://en.wikipedia.org/wiki/Comparison_of_audio_coding_formats. For textual data, there are almost 300 formats listed at http://fileinfo.com/filetypes/text. In this chapter, we will focus on how to download and extract these types of text as plain text for eventual processing.

We will briefly examine different data formats, followed by an examination of possible data sources. We need this knowledge to demonstrate how to obtain data using different data acquisition techniques.

Understanding the data formats used in data science applications

When we discuss data formats, we are referring to content format, as opposed to the underlying file format, which may not even be visible to most developers. We cannot examine all available formats due to the vast number of formats available. Instead, we will tackle several of the more common formats, providing adequate examples to address the most common data retrieval needs. Specifically, we will demonstrate how to retrieve data stored in the following formats:

- HTML
- PDF
- CSV/TSV
- Spreadsheets
- Databases
- JSON
- XML

Some of these formats are well supported and documented elsewhere. For example, XML has been in use for years and there are several well-established techniques for accessing XML data in Java. For these types of data, we will outline the major techniques available and show a few examples to illustrate how they work. This will provide those readers who are not familiar with the technology some insight into their nature.

The most common data format is binary files. For example, Word, Excel, and PDF documents are all stored in binary. These require special software to extract information from them. Text data is also very common.

Overview of CSV data

Comma Separated Values (**CSV**) files, contain tabular data organized in a row-column format. The data, stored as plaintext, is stored in rows, also called **records**. Each record contains fields separated by commas. These files are also closely related to other delimited files, most notably **Tab-Separated Values** (**TSV**) files. The following is a part of a simple CSV file, and these numbers are not intended to represent any specific type of data:

```
JURISDICTION NAME,COUNT PARTICIPANTS,COUNT FEMALE,PERCENT FEMALE
10001,44,22,0.5
10002,35,19,0.54
10003,1,1,1
```

Notice that the first row contains header data to describe the subsequent records. Each value is separated by a comma and corresponds to the header in the same position. In Chapter 3, *Data Cleaning*, we will discuss CSV files in more depth and examine the support available for different types of delimiters.

Overview of spreadsheets

Spreadsheets are a form of tabular data where information is stored in rows and columns, much like a two-dimensional array. They typically contain numeric and textual information and use formulas to summarize and analyze their contents. Most people are familiar with Excel spreadsheets, but they are also found as part of other product suites, such as OpenOffice.

Spreadsheets are an important data source because they have been used for the past several decades to store information in many industries and applications. Their tabular nature makes them easy to process and analyze. It is important to know how to extract data from this ubiquitous data source so that we can take advantage of the wealth of information that is stored in them.

For some of our examples, we will use a simple Excel spreadsheet that consists of a series of rows containing an ID, along with minimum, maximum, and average values. These numbers are not intended to represent any specific type of data. The spreadsheet looks like this:

ID	Minimum	Maximum	Average
12345	45	89	65.55
23456	78	96	86.75
34567	56	89	67.44
45678	86	99	95.67

In Chapter 3, *Data Cleaning*, we will learn how to extract data from spreadsheets.

Overview of databases

Data can be found in **Database Management Systems** (**DBMS**), which, like spreadsheets, are ubiquitous. Java provides a rich set of options for accessing and processing data in a DBMS. The intent of this section is to provide a basic introduction to database access using Java.

We will demonstrate the essence of connecting to a database, storing information, and retrieving information using JDBC. For this example, we used the MySQL DBMS. However, it will work for other DBMSes as well with a change in the database driver. We created a database called `example` and a table called `URLTABLE` using the following command within the **MySQL Workbench**. There are other tools that can achieve the same results:

```
CREATE TABLE IF NOT EXISTS `URLTABLE` (
  `RecordID` INT(11) NOT NULL AUTO_INCREMENT,
  `URL` text NOT NULL,
  PRIMARY KEY (`RecordID`)
);
```

We start with a `try` block to handle exceptions. A driver is needed to connect to the DBMS. In this example, we used `com.mysql.jdbc.Driver`. To connect to the database, the `getConnection` method is used, where the database server location, user ID, and password are passed. These values depend on the DBMS used:

```
    try {
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/example";
        connection = DriverManager.getConnection(url, "user ID",
            "password");
            ...
    } catch (SQLException | ClassNotFoundException ex) {
        // Handle exceptions
    }
```

Next, we will illustrate how to add information to the database and then how to read it. The SQL `INSERT` command is constructed in a string. The name of the MySQL database is `example`. This command will insert values into the `URLTABLE` table in the database where the question mark is a placeholder for the value to be inserted:

```
    String insertSQL = "INSERT INTO  `example`.`URLTABLE` "
        + "(`url`) VALUES " + "(?);";
```

The `PreparedStatement` class represents an SQL statement to execute. The `prepareStatement` method creates an instance of the class using the `INSERT` SQL statement:

```
    PreparedStatement stmt = connection.prepareStatement(insertSQL);
```

We then add URLs to the table using the `setString` method and the `execute` method. The `setString` method possesses two arguments. The first specifies the column index to insert the data and the second is the value to be inserted. The `execute` method does the actual insertion. We

add two URLs in the next sequence:

```
stmt.setString(1, "https://en.wikipedia.org/wiki/Data_science");
stmt.execute();
stmt.setString(1,
    "https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly");
stmt.execute();
```

To read the data, we use a SQL `SELECT` statement as declared in the `selectSQL` string. This will return all the rows and columns from the `URLTABLE` table. The `createStatement` method creates an instance of a `Statement` class, which is used for `INSERT` type statements. The `executeQuery` method executes the query and returns a `ResultSet` instance that holds the contents of the table:

```
String selectSQL = "select * from URLTABLE";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(selectSQL);
```

The following sequence iterates through the table, displaying one row at a time. The argument of the `getString` method specifies that we want to use the second column of the result set, which corresponds to the URL field:

```
out.println("List of URLs");
while (resultSet.next()) {
    out.println(resultSet.getString(2));
}
```

The output of this example, when executed, is as follows:

```
List of URLs
https://en.wikipedia.org/wiki/Data_science
https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly
```

If you need to empty the contents of the table, use the following sequence:

```
Statement statement = connection.createStatement();
statement.execute("TRUNCATE URLTABLE;");
```

This was a brief introduction to database access using Java. There are many resources available that will provide more in-depth coverage of this topic. For example, Oracle provides a more in-depth introduction to this topic at https://docs.oracle.com/javase/tutorial/jdbc/.

Overview of PDF files

The **Portable Document Format** (**PDF**) is a format not tied to a specific platform or software application. A PDF document can hold formatted text and images. PDF is an open standard, making it useful in a variety of places.

There are a large number of documents stored as PDF, making it a valuable source of data. There are several Java APIs that allow access to PDF documents, including **Apache POI** and **PDFBox**. Techniques for extracting information from a PDF document are illustrated in Chapter 3, *Data Cleaning*.

Overview of JSON

JavaScript Object Notation (**JSON**) (http://www.JSON.org/) is a data format used to interchange data. It is easy for humans or machines to read and write. JSON is supported by many languages, including Java, which has several JSON libraries listed at http://www.JSON.org/.

A JSON entity is composed of a set of name-value pairs enclosed in curly braces. We will use this format in several of our examples. In handling YouTube, we will use a JSON object, some of which is shown next, representing the results of a request from a YouTube video:

```
{
  "kind": "youtube#searchResult",
  "etag": "etag",
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  ...
}
```

Accessing the fields and values of such a document is not hard and is illustrated in Chapter 3, *Data Cleaning*.

Overview of XML

Extensible Markup Language (**XML**) is a markup language that specifies a standard document format. Widely used to communicate between applications and across the Internet, XML is popular due to its relative simplicity and flexibility. Documents encoded in XML are character-based and easily read by machines and humans.

XML documents contain markup and content characters. These characters allow parsers to classify the information contained within the document. The document consists of tags, and elements are stored within the tags. Elements may also contain other markup tags and form child elements. Additionally, elements may contain attributes or specific characteristics stored as a name-and-value pair.

An XML document must be well-formed. This means it must follow certain rules such as always using closing tags and only a single root tag. Other rules are discussed at https://en.wikipedia.org/wiki/XML#Well-formedness_and_error-handling.

The **Java API for XML Processing** (**JAXP**) consists of three interfaces for parsing XML data. The **Document Object Model** (**DOM**) interface parses an XML document and returns a tree structure delineating the structure of the document. The DOM interface parses an entire document as a whole. Alternatively, the **Simple API for XML** (**SAX**) parses a document one element at a time. SAX is preferable when memory usage is a concern as DOM requires more resources to construct the tree. DOM, however, offers flexibility over SAX in that any element can be accessed at any time and in any order.

The third Java API is known as **Streaming API for XML** (**StAX**). This streaming model was designed to accommodate the best parts of DOM and SAX models by granting flexibility without sacrificing resources. StAX exhibits higher performance, with the trade-off being that access is only available to one location in a document at a time. StAX is the preferred technique if you already know how you want to process the document, but it is also popular for applications with limited available memory.

The following is a simple XML file. Each `<text>` represents a tag, labelling the element contained within the tags. In this case, the largest node in our file is `<music>` and contained within it are sets of song data. Each tag within a `<song>` tag describes another element corresponding to that song. Every tag will eventually have a closing tag, such as `</song>`. Notice that the first tag contains information about which XML version should be used to parse the file:

```
<?xml version="1.0"?>
<music>
   <song id="1234">
      <artist>Patton, Courtney</artist>
      <name>So This Is Life</name>
      <genre>Country</genre>
      <price>2.99</price>
   </song>
   <song id="5678">
```

```
            <artist>Eady, Jason</artist>
            <name>AM Country Heaven</name>
            <genre>Country</genre>
            <price>2.99</price>
        </song>
</music>
```

There are numerous other XML-related technologies. For example, we can validate a specific XML document using either a DTD document or XML schema writing specifically for that XML document. XML documents can be transformed into a different format using XLST.

Overview of streaming data

Streaming data refers to data generated in a continuous stream and accessed in a sequential, piece-by-piece manner. Much of the data the average Internet user accesses is streamed, including video and audio channels, or text and image data on social media sites. Streaming data is the preferred method when the data is new and changing quickly, or when large data collections are sought.

Streamed data is often ideal for data science research because it generally exists in large quantities and raw format. Much public streaming data is available for free and supported by Java APIs. In this chapter, we are going to examine how to acquire data from streaming sources, including Twitter, Flickr, and YouTube. Despite the use of different techniques and APIs, you will notice similarities between the techniques used to pull data from these sites.

Overview of audio/video/images in Java

There are a large number of formats used to represent images, videos, and audio. This type of data is typically stored in binary format. Analog audio streams are sampled and digitized. Images are often simply collections of bits representing the color of a pixel. The following are links that provide a more in-depth discussion of some of these formats:

- Audio: https://en.wikipedia.org/wiki/Audio_file_format
- Images:https://en.wikipedia.org/wiki/Image_file_formats
- Video: https://en.wikipedia.org/wiki/Video_file_format

Frequently, this type of data can be quite large and must be compressed. When data is compressed two approaches are used. The first is a lossless compression, where less space is used and there is no loss of information. The second is lossy, where information is lost. Losing information is not always a bad thing as sometimes the loss is not noticeable to humans.

As we will demonstrate in Chapter 3, *Data Cleaning,* this type of data often is compromised in an inconvenient fashion and may need to be cleaned. For example, there may be background noise in an audio recording or an image may need to be smoothed before it can be processed. Image smoothing is demonstrated in Chapter 3, *Data Cleaning,* using the OpenCV library.

Data acquisition techniques

In this section, we will illustrate how to acquire data from web pages. Web pages contain a potential bounty of useful information. We will demonstrate how to access web pages using several technologies, starting with a low-level approach supported by the `HttpUrlConnection` class. To find pages, a web crawler application is often used. Once a useful page has been identified, then information needs to be extracted from the page. This is often performed using an HTML parser. Extracting this information is important because it is often buried amid a clutter of HTML tags and JavaScript code.

Using the HttpUrlConnection class

The contents of a web page can be accessed using the `HttpUrlConnection` class. This is a low-level approach that requires the developer to do a lot of footwork to extract relevant content. However, he or she is able to exercise greater control over how the content is handled. In some situations, this approach may be preferable to using other API libraries.

We will demonstrate how to download the content of Wikipedia's data science page using this class. We start with a `try/catch` block to handle exceptions. A URL object is created using the data science URL string. The `openConnection` method will create a connection to the Wikipedia server as shown here:

```
try {
    URL url = new URL(
        "https://en.wikipedia.org/wiki/Data_science");
    HttpURLConnection connection = (HttpURLConnection)
        url.openConnection();
    ...
} catch (MalformedURLException ex) {
    // Handle exceptions
} catch (IOException ex) {
    // Handle exceptions
}
```

The `connection` object is initialized with an HTTP `GET` command. The `connect` method is then executed to connect to the server:

```
connection.setRequestMethod("GET");
connection.connect();
```

Assuming no errors were encountered, we can determine whether the response was successful using the `getResponseCode` method. A normal return value is `200`. The content of a web page can vary. For example, the `getContentType` method returns a string describing the page's content. The `getContentLength` method returns its length:

```
out.println("Response Code: " + connection.getResponseCode());
out.println("Content Type: " + connection.getContentType());
out.println("Content Length: " + connection.getContentLength());
```

Assuming that we get an HTML formatted page, the next sequence illustrates how to get this content. A `BufferedReader` instance is created where one line at a time is read in from the web site and appended to a `BufferedReader` instance. The buffer is then displayed:

```
InputStreamReader isr = new InputStreamReader((InputStream)
    connection.getContent());
BufferedReader br = new BufferedReader(isr);
StringBuilder buffer = new StringBuilder();
String line;
do {
    line = br.readLine();
    buffer.append(line + "\n");
```

```
        } while (line != null);
        out.println(buffer.toString());
```

The abbreviated output is shown here:

```
Response Code: 200
Content Type: text/html; charset=UTF-8
Content Length: -1
<!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<meta charset="UTF-8"/>
<title>Data science - Wikipedia, the free encyclopedia</title>
<script>document.documentElement.className =
...
"wgHostname":"mw1251"});});</script>
</body>
</html>
```

While this is feasible, there are easier methods for getting the contents of a web page. One of these techniques is discussed in the next section.

Web crawlers in Java

Web crawling is the process of traversing a series of interconnected web pages and extracting relevant information from those pages. It does this by isolating and then following links on a page. While there are many precompiled datasets readily available, it may still be necessary to collect data directly off the Internet. Some sources such as news sites are continually being updated and need to be revisited from time to time.

A web crawler is an application that visits various sites and collects information. The web crawling process consists of a series of steps:

1. Select a URL to visit
2. Fetch the page
3. Parse the page
4. Extract relevant content
5. Extract relevant URLs to visit

This process is repeated for each URL visited.

There are several issues that need to be considered when fetching and parsing a page such as:

- **Page importance**: We do not want to process irrelevant pages.
- **Exclusively HTML**: We will not normally follow links to images, for example.
- **Spider traps**: We want to bypass sites that may result in an infinite number of requests. This can occur with dynamically generated pages where one request leads to another.
- **Repetition**: It is important to avoid crawling the same page more than once.
- **Politeness**: Do not make an excessive number of requests to a website. Observe the `robot.txt` files; they specify which parts of a site should not be crawled.

The process of creating a web crawler can be daunting. For all but the simplest needs, it is recommended that one of several open source web crawlers be used. A partial list follows:

- **Nutch**: http://nutch.apache.org
- **crawler4j**: https://github.com/yasserg/crawler4j
- **JSpider**: http://j-spider.sourceforge.net/
- **WebSPHINX**: http://www.cs.cmu.edu/~rcm/websphinx/
- **Heritrix**: https://webarchive.jira.com/wiki/display/Heritrix

We can either create our own web crawler or use an existing crawler and in this chapter we will examine both approaches. For specialized processing, it can be desirable to use a custom crawler. We will demonstrate how to create a simple web crawler in Java to provide more insight into how web crawlers work. This will be followed by a brief discussion of other web crawlers.

Creating your own web crawler

Now that we have a basic understanding of web crawlers, we are ready to create our own. In this simple web crawler, we will keep track of the pages visited using `ArrayList` instances. In addition, jsoup will be used to parse a web page and we will limit the number of pages we visit. Jsoup (https://jsoup.org/) is an open source HTML parser. This example demonstrates the basic structure of a web crawler and also highlights some of the issues involved in creating a web crawler.

We will use the `SimpleWebCrawler` class, as declared here:

```java
public class SimpleWebCrawler {

    private String topic;
    private String startingURL;
    private String urlLimiter;
    private final int pageLimit = 20;
    private ArrayList<String> visitedList = new ArrayList<>();
    private ArrayList<String> pageList = new ArrayList<>();
    ...
    public static void main(String[] args) {
        new SimpleWebCrawler();
    }

}
```

The instance variables are detailed here:

Variable	Use
topic	The keyword that needs to be in a page for the page to be accepted
startingURL	The URL of the first page
urlLimiter	A string that must be contained in a link before it will be followed
pageLimit	The maximum number of pages to retrieve
visitedList	The `ArrayList` containing pages that have already been visited
pageList	An `ArrayList` containing the URLs of the pages of interest

In the `SimpleWebCrawler` constructor, we initialize the instance variables to begin the search from the Wikipedia page for Bishop Rock, an island off the coast of Italy. This was chosen to minimize the number of pages that might be retrieved. As we will see, there are many more

Wikipedia pages dealing with Bishop Rock than one might think.

The `urlLimiter` variable is set to `Bishop_Rock`, which will restrict the embedded links to follow to just those containing that string. Each page of interest must contain the value stored in the `topic` variable. The `visitPage` method performs the actual crawl:

```
public SimpleWebCrawler() {
    startingURL = https://en.wikipedia.org/wiki/Bishop_Rock, "
        + "Isles_of_Scilly";
    urlLimiter = "Bishop_Rock";
    topic = "shipping route";
    visitPage(startingURL);
}
```

In the `visitPage` method, the `pageList` ArrayList is checked to see whether the maximum number of accepted pages has been exceeded. If the limit has been exceeded, then the search terminates:

```
public void visitPage(String url) {
    if (pageList.size() >= pageLimit) {
        return;
    }
    ...
}
```

If the page has already been visited, then we ignore it. Otherwise, it is added to the visited list:

```
if (visitedList.contains(url)) {
    // URL already visited
} else {
    visitedList.add(url);
        ...
}
```

`Jsoup` is used to parse the page and return a `Document` object. There are many different exceptions and problems that can occur such as a malformed URL, retrieval timeouts, or simply bad links. The `catch` block needs to handle these types of problems. We will provide a more in-depth explanation of jsoup in web scraping in Java:

```
try {
    Document doc = Jsoup.connect(url).get();
        ...
    }
} catch (Exception ex) {
    // Handle exceptions
}
```

If the document contains the topic text, then the link is displayed and added to the `pageList` ArrayList. Each embedded link is obtained, and if the link contains the limiting text, then the `visitPage` method is called recursively:

```
if (doc.text().contains(topic)) {
```

```
        out.println((pageList.size() + 1) + ": [" + url + "]");
        pageList.add(url);

        // Process page links
        Elements questions = doc.select("a[href]");
        for (Element link : questions) {
            if (link.attr("href").contains(urlLimiter)) {
                visitPage(link.attr("abs:href"));
            }
        }
    }
```

This approach only examines links in those pages that contain the topic text. Moving the `for` loop outside of the if statement will test the links for all pages.

The output follows:

```
1: [https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly]
2: [https://en.wikipedia.org/wiki/Bishop_Rock_Lighthouse]
3: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&oldid=717634231#Lighthouse]
4: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&diff=prev&oldid=717634231]
5: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&oldid=716622943]
6: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&diff=prev&oldid=716622943]
7: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&oldid=716608512]
8: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&diff=prev&oldid=716608512]
...
20: [https://en.wikipedia.org/w/index.php?
title=Bishop_Rock,_Isles_of_Scilly&diff=prev&oldid=716603919]
```

In this example, we did not save the results of the crawl in an external source. Normally this is necessary and can be stored in a file or database.

Using the crawler4j web crawler

Here we will illustrate the use of the crawler4j (https://github.com/yasserg/crawler4j) web crawler. We will use an adapted version of the basic crawler found at https://github.com/yasserg/crawler4j/tree/master/src/test/java/edu/uci/ics/crawler4j/examples/ba We will create two classes: `CrawlerController` and `SampleCrawler`. The former class set ups the crawler while the latter contains the logic that controls what pages will be processed.

As with our previous crawler, we will crawl the Wikipedia article dealing with Bishop Rock. The results using this crawler will be smaller as many extraneous pages are ignored.

Let's look at the `CrawlerController` class first. There are several parameters that are used with the crawler as detailed here:

- **Crawl storage folder**: The location where crawl data is stored
- **Number of crawlers**: This controls the number of threads used for the crawl
- **Politeness delay**: How many seconds to pause between requests
- **Crawl depth**: How deep the crawl will go
- **Maximum number of pages to fetch**: How many pages to fetch
- **Binary data**: Whether to crawl binary data such as PDF files

The basic class is shown here:

```java
public class CrawlerController {

  public static void main(String[] args) throws Exception {
    int numberOfCrawlers = 2;
    CrawlConfig config = new CrawlConfig();
    String crawlStorageFolder = "data";

    config.setCrawlStorageFolder(crawlStorageFolder);
    config.setPolitenessDelay(500);
    config.setMaxDepthOfCrawling(2);
    config.setMaxPagesToFetch(20);
    config.setIncludeBinaryContentInCrawling(false);
    ...
  }
}
```

Next, the `CrawlController` class is created and configured. Notice the `RobotstxtConfig` and `RobotstxtServer` classes used to handle `robot.txt` files. These files contain instructions that are intended to be read by a web crawler. They provide direction to help a crawler to do a better job such as specifying which parts of a site should not be crawled. This is useful for auto generated pages:

```java
PageFetcher pageFetcher = new PageFetcher(config);
RobotstxtConfig robotstxtConfig = new RobotstxtConfig();
RobotstxtServer robotstxtServer =
    new RobotstxtServer(robotstxtConfig, pageFetcher);
CrawlController controller =
    new CrawlController(config, pageFetcher, robotstxtServer);
```

The crawler needs to start at one or more pages. The `addSeed` method adds the starting pages. While we used the method only once here, it can be used as many times as needed:

```java
controller.addSeed(
  "https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly");
```

The `start` method will begin the crawling process:

```java
controller.start(SampleCrawler.class, numberOfCrawlers);
```

The `SampleCrawler` class contains two methods of interest. The first is the `shouldVisit` method that determines whether a page will be visited and the `visit` method that actually handles the page. We start with the class declaration and the declaration of a Java regular expression class

`Pattern` object. It will be one way of determining whether a page will be visited. In this declaration, standard images are specified and will be ignored:

```
public class SampleCrawler extends WebCrawler {
    private static final Pattern IMAGE_EXTENSIONS =
        Pattern.compile(".*\\.(bmp|gif|jpg|png)$");

    ...
}
```

The `shouldVisit` method is passed a reference to the page where this URL was found along with the URL. If any of the images match, the method returns `false` and the page is ignored. In addition, the URL must start with https://en.wikipedia.org/wiki/. We added this to restrict our searches to the Wikipedia website:

```
public boolean shouldVisit(Page referringPage, WebURL url) {
    String href = url.getURL().toLowerCase();
    if (IMAGE_EXTENSIONS.matcher(href).matches()) {
        return false;
    }
    return href.startsWith("https://en.wikipedia.org/wiki/");
}
```

The `visit` method is passed a `Page` object representing the page being visited. In this implementation, only those pages containing the string `shipping route` will be processed. This further restricts the pages visited. When we find such a page, its `URL`, `Text`, and `Text length` are displayed:

```
public void visit(Page page) {
    String url = page.getWebURL().getURL();

    if (page.getParseData() instanceof HtmlParseData) {
        HtmlParseData htmlParseData =
            (HtmlParseData) page.getParseData();
        String text = htmlParseData.getText();
        if (text.contains("shipping route")) {
            out.println("\nURL: " + url);
            out.println("Text: " + text);
            out.println("Text length: " + text.length());
        }
    }
}
```

The following is the truncated output of the program when executed:

```
URL: https://en.wikipedia.org/wiki/Bishop_Rock,_Isles_of_Scilly
Text: Bishop Rock, Isles of Scilly...From Wikipedia, the free encyclopedia
... Jump to: ... navigation, search For the Bishop Rock in the Pacific Ocean,
see Cortes Bank. Bishop Rock Bishop Rock Lighthouse (2005)
...
Text length: 14677
```

Notice that only one page was returned. This web crawler was able to identify and ignore

previous versions of the main web page.

We could perform further processing, but this example provides some insight into how the API works. Significant amounts of information can be obtained when visiting a page. In the example, we only used the URL and the length of the text. The following is a sample of other data that you may be interested in obtaining:

- URL path
- Parent URL
- Anchor
- HTML text
- Outgoing links
- Document ID

Web scraping in Java

Web scraping is the process of extracting information from a web page. The page is typically formatted using a series of HTML tags. An HTML parser is used to navigate through a page or series of pages and to access the page's data or metadata.

Jsoup (https://jsoup.org/) is an open source Java library that facilitates extracting and manipulating HTML documents using an HTML parser. It is used for a number of purposes, including web scraping, extracting specific elements from an HTML page, and cleaning up HTML documents.

There are several ways of obtaining an HTML document that may be useful. The HTML document can be extracted from a:

- URL
- String
- File

The first approach is illustrated next where the Wikipedia page for data science is loaded into a `Document` object. This `Jsoup` object represents the HTML document. The `connect` method connects to the site and the `get` method retrieves the `document`:

```
try {
    Document document = Jsoup.connect(
        "https://en.wikipedia.org/wiki/Data_science").get();
    ...
  } catch (IOException ex) {
    // Handle exception
}
```

Loading from a file uses the `File` class as shown next. The overloaded `parse` method uses the file to create the `document` object:

```
try {
    File file = new File("Example.html");
    Document document = Jsoup.parse(file, "UTF-8", "");
    ...
} catch (IOException ex) {
    // Handle exception
}
```

The `Example.html` file follows:

```
<html>
<head><title>Example Document</title></head>
<body>
<p>The body of the document</p>
Interesting Links:
<br>
<a href="https://en.wikipedia.org/wiki/Data_science">Data Science</a>
<br>
```

```
<a href="https://en.wikipedia.org/wiki/Jsoup">Jsoup</a>
<br>
Images:
<br>
 <img src="eyechart.jpg" alt="Eye Chart">
</body>
</html>
```

To create a `Document` object from a string, we will use the following sequence where the `parse` method processes the string that duplicates the previous HTML file:

```
String html = "<html>\n"
    + "<head><title>Example Document</title></head>\n"
    + "<body>\n"
    + "<p>The body of the document</p>\n"
    + "Interesting Links:\n"
    + "<br>\n"
    + "<a href=\"https://en.wikipedia.org/wiki/Data_science\">" +
      "DataScience</a>\n"
    + "<br>\n"
    + "<a href=\"https://en.wikipedia.org/wiki/Jsoup\">" +
      "Jsoup</a>\n"
    + "<br>\n"
    + "Images:\n"
    + "<br>\n"
    + " <img src=\"eyechart.jpg\" alt=\"Eye Chart\"> \n"
    + "</body>\n"
    + "</html>";
Document document = Jsoup.parse(html);
```

The `Document` class possesses a number of useful methods. The `title` method returns the title. To get the text contents of the document, the `select` method is used. This method uses a string specifying the element of a document to retrieve:

```
String title = document.title();
out.println("Title: " + title);
Elements element = document.select("body");
out.println("  Text: " + element.text());
```

The output for the Wikipedia data science page is shown here. It has been shortened to conserve space:

```
Title: Data science - Wikipedia, the free encyclopedia
Text: Data science From Wikipedia, the free encyclopedia Jump to: navigation,
search Not to be confused with information science. Part of a
...
policy About Wikipedia Disclaimers Contact Wikipedia Developers Cookie
statement Mobile view
```

The parameter type of the `select` method is a string. By using a string, the type of information selected is easily changed. Details on how to formulate this string are found at the jsoup Javadocs for the `Selector` class at https://jsoup.org/apidocs/:

We can use the `select` method to retrieve the images in a document, as shown here:

```
Elements images = document.select("img[src$=.png]");
for (Element image : images) {
    out.println("\nImage: " + image);
}
```

The output for the Wikipedia data science page is shown here. It has been shortened to conserve space:

```
Image: <img alt="Data Visualization" src="//upload.wikimedia.org/...>
Image: <img alt=""
src="//upload.wikimedia.org/wikipedia/commons/thumb/b/ba/...>
```

Links can be easily retrieved as shown next:

```
Elements links = document.select("a[href]");
for (Element link : links) {
    out.println("Link: " + link.attr("href")
        + " Text: " + link.text());
}
```

The output for the `Example.html` page is shown here:

```
Link: https://en.wikipedia.org/wiki/Data_science Text: Data Science
Link: https://en.wikipedia.org/wiki/Jsoup Text: Jsoup
```

jsoup possesses many additional capabilities. However, this example demonstrates the web scraping process. There are also other Java HTML parsers available. A comparison of Java HTML parser, among others, can be found at https://en.wikipedia.org/wiki/Comparison_of_HTML_parsers.

Using API calls to access common social media sites

Social media contain a wealth of information that can be processed and is used by many data analysis applications. In this section, we will illustrate how to access a few of these sources using their Java APIs. Most of them require some sort of access key, which is normally easy to obtain. We start with a discussion on the `OAuth` class, which provides one approach to authenticating access to a data source.

When working with the type of data source, it is important to keep in mind that the data is not always public. While it may be accessible, the owner of the data may be an individual who does not necessarily want the information shared. Most APIs provide a means to determine how the data can be distributed, and these requests should be honored. When private information is used, permission from the author must be obtained.

In addition, these sites have limits on the number of requests that can be made. Keep this in mind when pulling data from a site. If these limits need to be exceeded, then most sites provide a way of doing this.

Using OAuth to authenticate users

OAuth is an open standard used to authenticate users to many different websites. A resource owner effectively delegates access to a server resource without having to share their credentials. It works over HTTPS. OAuth 2.0 succeeded OAuth and is not backwards compatible. It provides client developers a simple way of providing authentication. Several companies use OAuth 2.0 including PayPal, Comcast, and Blizzard Entertainment.

A list of OAuth 2.0 providers is found at https://en.wikipedia.org/wiki/List_of_OAuth_providers. We will use several of these in our discussions.

Handing Twitter

The sheer volume of data and the popularity of the site, among celebrities and the general public alike, make Twitter a valuable resource for mining social media data. Twitter is a popular social media platform allowing users to read and post short messages called **tweets**. Twitter provides API support for posting and pulling tweets, including streaming data from all public users. While there are services available for pulling the entire set of public tweet data, we are going to examine other options that, while limiting in the amount of data retrieved at one time, are available at no cost.

We are going to focus on the Twitter API for retrieving streaming data. There are other options for retrieving tweets from a specific user as well as posting data to a specific account but we will not be addressing those in this chapter. The public stream API, at the default access level, allows the user to pull a sample of public tweets currently streaming on Twitter. It is possible to refine the data by specifying parameters to track keywords, specific users, and location.

We are going to use HBC, a Java HTTP client, for this example. You can download a sample HBC application at https://github.com/twitter/hbc. If you prefer to use a different HTTP client, ensure it will return incremental response data. The Apache HTTP client is one option. Before you can create the HTTP connection, you must first create a Twitter account and an application within that account. To get started with the app, visit apps.twitter.com. Once your app is created, you will be assigned a consumer key, consumer secret, access token, and access secret token. We will also use OAuth, as discussed previously in this chapter.

First, we will write a method to perform the authentication and request data from Twitter. The parameters for our method are the authentication information given to us by Twitter when we created our app. We will create a `BlockingQueue` object to hold our streaming data. For this example, we will set a default capacity of 10,000. We will also specify our endpoint and turn off stall warnings:

```
public static void streamTwitter(
    String consumerKey, String consumerSecret,
    String accessToken, String accessSecret)
        throws InterruptedException {

    BlockingQueue<String> statusQueue =
        new LinkedBlockingQueue<String>(10000);
    StatusesSampleEndpoint ending =
        new StatusesSampleEndpoint();
    ending.stallWarnings(false);
    ...
}
```

Next, we create an `Authentication` object using `OAuth1`, a variation of the `OAuth` class. We can then build our connection client and complete the HTTP connection:

```
Authentication twitterAuth = new OAuth1(consumerKey,
    consumerSecret, accessToken, accessSecret);
BasicClient twitterClient = new ClientBuilder()
        .name("Twitter client")
        .hosts(Constants.STREAM_HOST)
        .endpoint(ending)
        .authentication(twitterAuth)
        .processor(new StringDelimitedProcessor(statusQueue))
        .build();
twitterClient.connect();
```

For the purposes of this example, we will simply read the messages received from the stream and print them to the screen. The messages are returned in JSON format and the decision of how to process them in a real application will depend upon the purpose and limitations of that application:

```
for (int msgRead = 0; msgRead < 1000; msgRead++) {
  if (twitterClient.isDone()) {
    out.println(twitterClient.getExitEvent().getMessage());
    break;
  }
```

```
    String msg = statusQueue.poll(10, TimeUnit.SECONDS);
    if (msg == null) {
      out.println("Waited 10 seconds - no message received");
    } else {
      out.println(msg);
    }
  }
  twitterClient.stop();
```

To execute our method, we simply pass our authentication information to the `streamTwitter` method. For security purposes, we have replaced our personal keys here. Authentication information should always be protected:

```
public static void main(String[] args) {

  try {
    SampleStreamExample.streamTwitter(
        myKey, mySecret, myToken, myAccess);
  } catch (InterruptedException e) {
    out.println(e);
  }
}
```

Here is truncated sample data retrieved using the methods listed above. Your data will vary based upon Twitter's live stream, but it should resemble this example:

{"created_at":"Fri May 20 15:47:21 +0000 2016","id":733685552789098496,"id_str":"733685552789098496","text":"bwisit si em bahala sya","source":"\u003ca href="http:\/\/twitter.com" rel="nofollow"\u003eTwitter Web ... ntions":[],"symbols": []},"favorited":false,"retweeted":false,"filter_level":"low","lang":"tl","tim estamp_ms":"1463759241660"}

Twitter also provides support for pulling all data for one specific user account, as well as posting data directly to an account. A REST API is also available and provides support for specific queries via the search API. These also use the OAuth standard and return data in JSON files.

Handling Wikipedia

Wikipedia (https://www.wikipedia.org/) is a useful source of text and image type information. It is an Internet encyclopedia that hosts 38 million articles written in over 250 languages (https://en.wikipedia.org/wiki/Wikipedia). As such, it is useful to know how to programmatically access its contents.

MediaWiki is an open source wiki application that supports wiki type sites. It is used to support Wikipedia and many other sites. The MediaWiki API (http://www.mediawiki.org/wiki/API) provides access to a wiki's data and metadata over HTTP. An application, using this API, can log in, read data, and post changes to a site.

There are several Java APIs that support programmatic access to a wiki site as listed at https://www.mediawiki.org/wiki/API:Client_code#Java. To demonstrate Java access to a wiki we will use Bliki found at https://bitbucket.org/axelclk/info.bliki.wiki/wiki/Home. It provides good access and is easy to use for most basic operations.

The MediaWiki API is complex and has many features. The intent of this section is to illustrate the basic process of obtaining text from a Wikipedia article using this API. It is not possible to cover the API completely here.

We will use the following classes from the `info.bliki.api` and `info.bliki.wiki.model` packages:

- `Page`: Represents a retrieved page
- `User`: Represents a user
- `WikiModel`: Represents the wiki

Javadocs for Bliki are found at http://www.javadoc.io/doc/info.bliki.wiki/bliki-core/3.1.0.

The following example has been adapted from http://www.integratingstuff.com/2012/04/06/hook-into-wikipedia-using-java-and-the-mediawiki-api/. This example will access the English Wikipedia page for the subject, data science. We start by creating an instance of the `User` class. The first two arguments of the three-argument constructor are the `user ID` and `password`, respectively. In this case, they are empty strings. This combination allows us to read a page without having to set up an account. The third argument is the URL for the MediaWiki API page:

```
User user = new User("", "",
    "http://en.wikipedia.org/w/api.php");
user.login();
```

An account will enable us to modify the document. The `queryContent` method returns a list of `Page` objects for the subjects found in a string array. Each string should be the title of a page. In this example, we access a single page:

```
String[] titles = {"Data science"};
List<Page> pageList = user.queryContent(titles);
```

Each `Page` object contains the content of a page. There are several methods that will return the contents of the page. For each page, a `WikiModel` instance is created using the two-argument constructor. The first argument is the image base URL and the second argument is the link base URL. These URLs use Wiki variables called `image` and `title`, which will be replaced when creating links:

```
for (Page page : pageList) {
    WikiModel wikiModel = new WikiModel("${image}",
        "${title}");
    ...
}
```

The `render` method will take the wiki page and render it to HTML. There is also a method to

render the page to a PDF document:

```
String htmlText = wikiModel.render(page.toString());
```

The HTML text is then displayed:

```
out.println(htmlText);
```

A partial listing of the output follows:

```
<p>PageID: 35458904; NS: 0; Title: Data science;
Image url:
Content:
{{distinguish}}
{{Use dmy dates}}
{{Data Visualization}}</p>
<p><b>Data science</b> is an interdisciplinary field about processes and
systems to extract <a href="Knowledge" >knowledge</a>
...
```

We can also obtain basic information about the article using one of several methods as shown here:

```
out.println("Title: " + page.getTitle() + "\n" +
    "Page ID: " + page.getPageid() + "\n" +
    "Timestamp: " + page.getCurrentRevision().getTimestamp());
```

It is also possible to obtain a list of references in the article and a list of the headers. Here, a list of the references is displayed:

```
List <Reference> referenceList = wikiModel.getReferences();
out.println(referenceList.size());
for(Reference reference : referenceList) {
    out.println(reference.getRefString());
}
```

The following illustrates the process of getting the section headers:

```
ITableOfContent toc = wikiModel.getTableOfContent();
List<SectionHeader> sections = toc.getSectionHeaders();
for(SectionHeader sh : sections) {
    out.println(sh.getFirst());
}
```

The entire content of Wikipedia can be downloaded. This process is discussed at https://en.wikipedia.org/wiki/Wikipedia:Database_download.

It may be desirable to set up your own Wikipedia server to handle your request.

Handling Flickr

Flickr (https://www.flickr.com/) is an online photo management and sharing application. It is a possible source for images and videos. The Flickr Developer Guide

(https://www.flickr.com/services/developer/) is a good starting point to learn more about Flickr's API.

One of the first steps to using the Flickr API is to request an API key. This key is used to sign your API requests. The process to obtain a key starts at https://www.flickr.com/services/apps/create/. Both commercial and noncommercial keys are available. When you obtain a key you will also get a "secret." Both of these are required to use the API.

We will illustrate the process of locating and downloading images from Flickr. The process involves:

- Creating a Flickr class instance
- Specifying the search parameters for a query
- Performing the search
- Downloading the image

A `FlickrException` or `IOException` may be thrown during this process. There are several APIs that support Flickr access. We will be using Flickr4Java, found at https://github.com/callmeal/Flickr4Java. The Flickr4Java Javadocs is found at http://flickrj.sourceforge.net/api/. We will start with a `try` block and the `apikey` and `secret` declarations:

```
try {
    String apikey = "Your API key";
    String secret = "Your secret";

} catch (FlickrException | IOException ex) {
    // Handle exceptions
}
```

The `Flickr` instance is created next, where the `apikey` and `secret` are supplied as the first two parameters. The last parameter specifies the transfer technique used to access Flickr servers. Currently, the REST transport is supported using the `REST` class:

```
Flickr flickr = new Flickr(apikey, secret, new REST());
```

To search for images, we will use the `SearchParameters` class. This class supports a number of criteria that will narrow down the number of images returned from a query and includes such criteria as latitude, longitude, media type, and user ID. In the following sequence, the `setBBox` method specifies the longitude and latitude for the search. The parameters are (in order): minimum longitude, minimum latitude, maximum longitude, and maximum latitude. The `setMedia` method specifies the type of media. There are three possible arguments — `"all"`, `"photos"`, and `"videos"`:

```
SearchParameters searchParameters = new SearchParameters();
searchParameters.setBBox("-180", "-90", "180", "90");
searchParameters.setMedia("photos");
```

The `PhotosInterface` class possesses a `search` method that uses the `SearchParameters`

instance to retrieve a list of photos. The `getPhotosInterface` method returns an instance of the `PhotosInterface` class, as shown next. The `SearchParameters` instance is the first parameter. The second parameter determines how many photos are retrieved per page and the third parameter is the offset. A `PhotoList` class instance is returned:

```
PhotosInterface pi = new PhotosInterface(apikey, secret,
    new REST());
PhotoList<Photo> list = pi.search(searchParameters, 10, 0);
```

The next sequence illustrates the use of several methods to get information about the images retrieved. Each `Photo` instance is accessed using the `get` method. The title, image format, public flag, and photo URL are displayed:

```
out.println("Image List");
for (int i = 0; i < list.size(); i++) {
    Photo photo = list.get(i);
    out.println("Image: " + i +
        `"\nTitle: " + photo.getTitle() +
        "\nMedia: " + photo.getOriginalFormat() +
        "\nPublic: " + photo.isPublicFlag() +
        "\nUrl: " + photo.getUrl() +
        "\n");
}
out.println();
```

A partial listing is shown here where many of the specific values have been modified to protect the original data:

```
Image List
Image: 0
Title: XYZ Image
Media: jpg
Public: true
Url: https://flickr.com/photos/7723...@N02/269...
Image: 1
Title: IMG_5555.jpg
Media: jpg
Public: true
Url: https://flickr.com/photos/2665...@N07/264...
Image: 2
Title: DSC05555
Media: jpg
Public: true
Url: https://flickr.com/photos/1179...@N04/264...
```

The list of images returned by this example will vary since we used a fairly wide search range and images are being added all of the time.

There are two approaches that we can use to download an image. The first uses the image's URL and the second uses a `Photo` object. The image's URL can be obtained from a number of sources. We use the `Photo` class `getUrl` method for this example.

In the following sequence, we obtain an instance of `PhotosInterface` using its constructor to illustrate an alternate approach:

```
PhotosInterface pi = new PhotosInterface(apikey, secret,
    new REST());
```

We get the first `Photo` instance from the previous list and then its `getUrl` to get the image's URL. The `PhotosInterface` class's `getImage` method returns a `BufferedImage` object representing the image as shown here:

```
Photo currentPhoto = list.get(0);
BufferedImage bufferedImage =
    pi.getImage(currentPhoto.getUrl());
```

The image is then saved to a file using the `ImageIO` class:

```
File outputfile = new File("image.jpg");
ImageIO.write(bufferedImage, "jpg", outputfile);
```

The `getImage` method is overloaded. Here, the `Photo` instance and the size of the image desired are used as arguments to get the `BufferedImage` instance:

```
bufferedImage = pi.getImage(currentPhoto, Size.SMALL);
```

The image can be saved to a file using the previous technique.

The Flickr4Java API supports a number of other techniques for working with Flickr images.

Handling YouTube

YouTube is a popular video site where users can upload and share videos (https://www.youtube.com/). It has been used to share humorous videos, provide instructions on how to do any number of things, and share information among its viewers. It is a useful source of information as it captures the thoughts and ideas of a diverse group of people. This provides an interesting opportunity to analysis and gain insight into human behavior.

YouTube can serve as a useful source of videos and video metadata. A Java API is available to access its contents (https://developers.google.com/youtube/v3/). Detailed documentation of the API is found at https://developers.google.com/youtube/v3/docs/.

In this section, we will demonstrate how to search for videos by keyword and retrieve information of interest. We will also show how to download a video. To use the YouTube API, you will need a Google account, which can be obtained at https://www.google.com/accounts/NewAccount. Next, create an account in the Google Developer's Console (https://console.developers.google.com/). API access is supported using either API keys or OAuth 2.0 credentials. The project creation process and keys are discussed at https://developers.google.com/youtube/registering_an_application#create_project.

Searching by keyword

The process of searching for videos by keyword is adapted from
https://developers.google.com/youtube/v3/code_samples/java#search_by_keyword. Other
potentially useful code examples can be found at
https://developers.google.com/youtube/v3/code_samples/java. The process has been simplified
so that we can focus on the search process. We start with a try block and the creation of a
`YouTube` instance. This class provides the basic access to the API. Javadocs for this API is found
at https://developers.google.com/resources/api-libraries/documentation/youtube/v3/java/latest/.

The `YouTube.Builder` class is used to construct a `YouTube` instance. Its constructor takes three
arguments:

- `Transport`: Object used for HTTP
- `JSONFactory`: Used to process JSON objects
- `HttpRequestInitializer`: None is needed for this example

Many of the APIs responses will be in the form of JSON objects. The YouTube class'
`setApplicationName` method gives it a name and the `build` method creates a new YouTube
instance:

```
try {
    YouTube youtube = new YouTube.Builder(
        Auth.HTTP_TRANSPORT,
        Auth.JSON_FACTORY,
        new HttpRequestInitializer() {
            public void initialize(HttpRequest request)
                    throws IOException {
            }
        })
            .setApplicationName("application_name")
    ...
} catch (GoogleJSONResponseException ex) {
    // Handle exceptions
} catch (IOException ex) {
    // Handle exceptions
}
```

Next, we initialize a string to hold the search term of interest. In this case, we will look for
videos containing the word `cats`:

```
String queryTerm = "cats";
```

The class, `YouTube.Search.List`, maintains a collection of search results. The `YouTube` class's
`search` method specifies the type of resource to be returned. In this case, the string specifies the
`id` and `snippet` portions of the search result to be returned:

```
YouTube.Search.List search = youtube
    .search()
    .list("id,snippet");
```

The search result is a JSON object that has the following structure. It is described in more detail
at https://developers.google.com/youtube/v3/docs/playlistItems#methods. In the previous

sequence, only the `id` and `snippet` parts of a search will be returned, resulting in a more efficient operation:

```
{
  "kind": "youtube#searchResult",
  "etag": etag,
  "id": {
    "kind": string,
    "videoId": string,
    "channelId": string,
    "playlistId": string
  },
  "snippet": {
    "publishedAt": datetime,
    "channelId": string,
    "title": string,
    "description": string,
    "thumbnails": {
      (key): {
        "url": string,
        "width": unsigned integer,
        "height": unsigned integer
      }
    },
    "channelTitle": string,
    "liveBroadcastContent": string
  }
}
```

Next, we need to specify the API key and various search parameters. The query term is specified, as well as the type of media to be returned. In this case, only videos will be returned. The other two options include `channel` and `playlist`:

```
String apiKey = "Your API key";
search.setKey(apiKey);
search.setQ(queryTerm);
search.setType("video");
```

In addition, we further specify the fields to be returned as shown here. These correspond to fields of the JSON object:

```
search.setFields("items(id/kind,id/videoId,snippet/title," +
    "snippet/description,snippet/thumbnails/default/url)");
```

We also specify the maximum number of results to retrieve using the `setMaxResults` method:

```
search.setMaxResults(10L);
```

The `execute` method will perform the actual query, returning a `SearchListResponse` object. Its `getItems` method returns a list of `SearchResult` objects, one for each video retrieved:

```
SearchListResponse searchResponse = search.execute();
List<SearchResult> searchResultList =
    searchResponse.getItems();
```

In this example, we do not iterate through each video returned. Instead, we retrieve the first video and display information about the video. The `SearchResult` video variable allows us to access different parts of the JSON object, as shown here:

```
SearchResult video = searchResultList.iterator().next();
Thumbnail thumbnail = video
    .getSnippet().getThumbnails().getDefault();

out.println("Kind: " + video.getKind());
out.println("Video Id: " + video.getId().getVideoId());
out.println("Title: " + video.getSnippet().getTitle());
out.println("Description: " +
    video.getSnippet().getDescription());
out.println("Thumbnail: " + thumbnail.getUrl());
```

One possible output follows where parts of the output have been modified:

```
Kind: null
Video Id: tntO...
Title: Funny Cats ...
Description: Check out the ...
Thumbnail: https://i.ytimg.com/vi/tntO.../default.jpg
```

We have skipped many error checking steps to simplify the example, but these should be considered when implementing this in a business application.

If we need to download the video, one of the simplest ways is to use axet/wget found at https://github.com/axet/wget. It provides an easy-to-use technique to download the video using its video ID.

In the following example, a URL is created using the video ID. You will need to provide a video ID for this to work properly. The file is saved to the current directory with the video's title as the filename:

```
String url = "http://www.youtube.com/watch?v=videoID";
String path = ".";
VGet vget = new VGet(new URL(url), new File(path));
vget.download();
```

There are other more sophisticated download techniques found at the GitHub site.

Summary

In this chapter, we discussed types of data that are useful for data science and readily accessible on the Internet. This discussion included details about file specifications and formats for the most common types of data sources.

We also examined Java APIs and other techniques for retrieving data, and illustrated this process with multiple sources. In particular, we focused on types of text-based document formats and multimedia files. We used web crawlers to access websites and then performed web scraping to retrieve data from the sites we encountered.

Finally, we extracted data from social media sites and examined the available Java support. We retrieved data from Twitter, Wikipedia, Flickr, and YouTube and examined the available API support.

Chapter 3. Data Cleaning

Real-world data is frequently dirty and unstructured, and must be reworked before it is usable. Data may contain errors, have duplicate entries, exist in the wrong format, or be inconsistent. The process of addressing these types of issues is called **data cleaning**. Data cleaning is also referred to as **data wrangling**, **massaging**, **reshaping** , or **munging**. Data merging, where data from multiple sources is combined, is often considered to be a data cleaning activity.

We need to clean data because any analysis based on inaccurate data can produce misleading results. We want to ensure that the data we work with is quality data. Data quality involves:

- **Validity**: Ensuring that the data possesses the correct form or structure
- **Accuracy:** The values within the data are truly representative of the dataset
- **Completeness:** There are no missing elements
- **Consistency**: Changes to data are in sync
- **Uniformity**: The same units of measurement are used

There are several techniques and tools used to clean data. We will examine the following approaches:

- Handling different types of data
- Cleaning and manipulating text data
- Filling in missing data
- Validating data

In addition, we will briefly examine several image enhancement techniques.

There are often many ways to accomplish the same cleaning task. For example, there are a number of GUI tools that support data cleaning, such as OpenRefine (http://openrefine.org/). This tool allows a user to read in a dataset and clean it using a variety of techniques. However, it requires a user to interact with the application for each dataset that needs to be cleaned. It is not conducive to automation.

We will focus on how to clean data using Java code. Even then, there may be different techniques to clean the data. We will show multiple approaches to provide the reader with insights on how it can be done. Sometimes, this will use core Java string classes, and at other time, it may use specialized libraries.

These libraries often are more expressive and efficient. However, there are times when using a simple string function is more than adequate to address the problem. Showing complimentary techniques will improve the reader's skill set.

The basic text based tasks include:

- Data transformation
- Data imputation (handling missing data)

- Subsetting data
- Sorting data
- Validating data

In this chapter, we are interested in cleaning data. However, part of this process is extracting information from various data sources. The data may be stored in plaintext or in binary form. We need to understand the various formats used to store data before we can begin the cleaning process. Many of these formats were introduced in Chapter 2, *Data Acquisition,* but we will go into greater detail in the following sections.

Handling data formats

Data comes in all types of forms. We will examine the more commonly used formats and show how they can be extracted from various data sources. Before we can clean data it needs to be extracted from a data source such as a file. In this section, we will build upon the introduction to data formats found in Chapter 2, *Data Acquisition*, and show how to extract all or part of a dataset. For example, from an HTML page we may want to extract only the text without markup. Or perhaps we are only interested in its figures.

These data formats can be quite complex. The intent of this section is to illustrate the basic techniques commonly used with that data format. Full treatment of a specific data format is beyond the scope of this book. Specifically, we will introduce how the following data formats can be processed from Java:

- CSV data
- Spreadsheets
- Portable Document Format, or PDF files
- Javascript Object Notation, or JSON files

There are many other file types not addressed here. For example, jsoup is useful for parsing HTML documents. Since we introduced how this is done in the Web scraping in Java section of Chapter 2, *Data Acquisition*, we will not duplicate the effort here.

Handling CSV data

A common technique for separating information is to use commas or similar separators. Knowing how to work with CSV data allows us to utilize this type of data in our analysis efforts. When we deal with CSV data there are several issues including escaped data and embedded commas.

We will examine a few basic techniques for processing comma-separated data. Due to the row-column structure of CSV data, these techniques will read data from a file and place the data in a two-dimensional array. First, we will use a combination of the `Scanner` class to read in tokens and the `String` class `split` method to separate the data and store it in the array. Next, we will explore using the third-party library, OpenCSV, which offers a more efficient technique.

However, the first approach may only be appropriate for quick and dirty processing of data. We will discuss each of these techniques since they are useful in different situations.

We will use a dataset downloaded from https://www.data.gov/ containing U.S. demographic statistics sorted by ZIP code. This dataset can be downloaded at https://catalog.data.gov/dataset/demographic-statistics-by-zip-code-acfc9. For our purposes, this dataset has been stored in the file `Demographics.csv`. In this particular file, every row contains the same number of columns. However, not all data will be this clean and the solutions shown next take into account the possibility for jagged arrays.

Note

A jagged array is an array where the number of columns may be different for different rows. For example, row 2 may have 5 elements while row 3 may have 6 elements. When using jagged arrays you have to be careful with your column indexes.

First, we use the `Scanner` class to read in data from our data file. We will temporarily store the data in an `ArrayList` since we will not always know how many rows our data contains.

```
try (Scanner csvData = new Scanner(new File("Demographics.csv"))) {
    ArrayList<String> list = new ArrayList<String>();
    while (csvData.hasNext()) {
        list.add(csvData.nextLine());
} catch (FileNotFoundException ex) {
    // Handle exceptions
}
```

The list is converted to an array using the `toArray` method. This version of the method uses a `String` array as an argument so that the method will know what type of array to create. A two-dimension array is then created to hold the CSV data.

```
String[] tempArray = list.toArray(new String[1]);
String[][] csvArray = new String[tempArray.length][];
```

The `split` method is used to create an array of `Strings` for each row. This array is assigned to a row of the `csvArray`.

```
for(int i=0; i<tempArray.length; i++) {
    csvArray[i] = tempArray[i].split(",");
}
```

Our next technique will use a third-party library to read in and process CSV data. There are multiple options available, but we will focus on the popular OpenCSV (http://opencsv.sourceforge.net). This library offers several advantages over our previous technique. We can have an arbitrary number of items on each row without worrying about handling exceptions. We also do not need to worry about embedded commas or embedded carriage returns within the data tokens. The library also allows us to choose between reading the entire file at once or using an iterator to process data line-by-line.

First, we need to create an instance of the `CSVReader` class. Notice the second parameter allows us to specify the delimiter, a useful feature if we have similar file format delimited by tabs or dashes, for example. If we want to read the entire file at one time, we use the `readAll` method.

```
CSVReader dataReader = new CSVReader(new
FileReader("Demographics.csv"),',');
ArrayList<String> holdData = (ArrayList)dataReader.readAll();
```

We can then process the data as we did above, by splitting the data into a two-dimension array using `String` class methods. Alternatively, we can process the data one line at a time. In the example that follows, each token is printed out individually but the tokens can also be stored in a two-dimension array or other data structure as appropriate.

```
CSVReader dataReader = new CSVReader(new
FileReader("Demographics.csv"),',');
String[] nextLine;
while ((nextLine = dataReader.readNext()) != null){
for(String token : nextLine){
    out.println(token);
  }
}
dataReader.close();
```

We can now clean or otherwise process the array.

Handling spreadsheets

Spreadsheets have proven to be a very popular tool for processing numeric and textual data. Due to the wealth of information that has been stored in spreadsheets over the past decades, knowing how to extract information from spreadsheets enables us to take advantage of this widely available data source. In this section, we will demonstrate how this is accomplished using the Apache POI API.

Open Office also supports a spreadsheet application. Open Office documents are stored in XML format which makes it readily accessible using XML parsing technologies. However, the Apache ODF Toolkit (http://incubator.apache.org/odftoolkit/) provides a means of accessing data within a document without knowing the format of the OpenOffice document. This is currently an incubator project and is not fully mature. There are a number of other APIs that can assist in processing OpenOffice documents as detailed on the **Open Document Format** (**ODF**) for developers (http://www.opendocumentformat.org/developers/) page.

Handling Excel spreadsheets

Apache POI (http://poi.apache.org/index.html) is a set of APIs providing access to many Microsoft products including Excel and Word. It consists of a series of components designed to access a specific Microsoft product. An overview of these components is found at http://poi.apache.org/overview.html.

In this section we will demonstrate how to read a simple Excel spreadsheet using the XSSF component to access Excel 2007+ spreadsheets. The Javadocs for the Apache POI API is found at https://poi.apache.org/apidocs/index.html.

We will use a simple Excel spreadsheet consisting of a series of rows containing an ID along with minimum, maximum, and average values. These numbers are not intended to represent any specific type of data. The spreadsheet follows:

ID	Minimum	Maximum	Average
12345	45	89	65.55
23456	78	96	86.75
34567	56	89	67.44
45678	86	99	95.67

We start with a try-with-resources block to handle any `IOExceptions` that may occur:

```
try (FileInputStream file = new FileInputStream(
        new File("Sample.xlsx"))) {
    ...
    }
} catch (IOException e) {
    // Handle exceptions
}
```

An instance of a XSSFWorkbook class is created using the spreadsheet. Since a workbook may consists of multiple spreadsheets, we select the first one using the getSheetAt method.

```
XSSFWorkbook workbook = new XSSFWorkbook(file);
XSSFSheet sheet = workbook.getSheetAt(0);
```

The next step is to iterate through the rows, and then each column, of the spreadsheet:

```
for(Row row : sheet) {
    for (Cell cell : row) {
        ...
    }
}
out.println();
```

Each cell of the spreadsheet may use a different format. We use the getCellType method to determine its type and then use the appropriate method to extract the data in the cell. In this example we are only working with numeric and text data.

```
switch (cell.getCellType()) {
    case Cell.CELL_TYPE_NUMERIC:
        out.print(cell.getNumericCellValue() + "\t");
        break;
    case Cell.CELL_TYPE_STRING:
        out.print(cell.getStringCellValue() + "\t");
        break;
    }
```

When executed we get the following output:

```
ID Minimum Maximum Average
12345.0 45.0 89.0 65.55
23456.0 78.0 96.0 86.75
34567.0 56.0 89.0 67.44
45678.0 86.0 99.0 95.67
```

POI supports other more sophisticated classes and methods to extract data.

Handling PDF files

There are several APIs supporting the extraction of text from a PDF file. Here we will use PDFBox. The Apache PDFBox (https://pdfbox.apache.org/) is an open source API that allows Java programmers to work with PDF documents. In this section we will illustrate how to extract simple text from a PDF document. Javadocs for the PDFBox API is found at https://pdfbox.apache.org/docs/2.0.1/javadocs/.

This is a simple PDF file. It consists of several bullets:

- Line 1
- Line 2
- Line 3

This is the end of the document.

A `try` block is used to catch `IOExceptions`. The `PDDocument` class will represent the PDF document being processed. Its `load` method will load in the PDF file specified by the `File` object:

```
try {
    PDDocument document = PDDocument.load(new File("PDF File.pdf"));
    ...
} catch (Exception e) {
    // Handle exceptions
}
```

Once loaded, the `PDFTextStripper` class `getText` method will extract the text from the file. The text is then displayed as shown here:

```
PDFTextStripper Tstripper = new PDFTextStripper();
String documentText = Tstripper.getText(document);
System.out.println(documentText);
```

The output of this example follows. Notice that the bullets are returned as question marks.

```
This is a simple PDF file. It consists of several bullets:
? Line 1
? Line 2
? Line 3
This is the end of the document.
```

This is a brief introduction to the use of PDFBox. It is a very powerful tool when we need to extract and otherwise manipulate PDF documents.

Handling JSON

In [Chapter 2](#), *Data Acquisition* we learned that certain YouTube searches return JSON formatted results. Specifically, the `SearchResult` class holds information relating to a specific search. In that section we illustrate how to use YouTube specific techniques to extract information. In this section we will illustrate how to extract JSON information using the Jackson JSON implementation.

JSON supports three models for processing data:

- **Streaming API** - JSON data is processed token by token
- **Tree model** - The JSON data is held entirely in memory and then processed
- **Data binding** - The JSON data is transformed to a Java object

Using JSON streaming API

We will illustrate the first two approaches. The first approach is more efficient and is used when a large amount of data is processed. The second technique is convenient but the data must not be too large. The third technique is useful when it is more convenient to use specific Java classes to process data. For example, if the JSON data represent an address then a specific Java address class cane be defined to hold and process the data.

There are several Java libraries that support JSON processing including:

- Flexjson (http://flexjson.sourceforge.net/)
- Genson (http://owlike.github.io/genson/)
- Google-Gson (https://github.com/google/gson)
- Jackson library (https://github.com/FasterXML/jackson)
- JSON-io (https://github.com/jdereg/json-io)
- JSON-lib (http://json-lib.sourceforge.net/)

We will use the Jackson Project (https://github.com/FasterXML/jackson). Documentation is found at https://github.com/FasterXML/jackson-docs. We will use two JSON files to demonstrate how it can be used. The first file, `Person.json`, is shown next where a single person data is stored. It consists of four fields where the last field is an array of location information.

```
{
   "firstname":"Smith",
   "lastname":"Peter",
   "phone":8475552222,
   "address":["100 Main Street","Corpus","Oklahoma"]
}
```

The code sequence that follows shows how to extract the values for each of the fields. Within the try-catch block a `JsonFactory` instance is created which then creates a `JsonParser` instance based on the `Person.json` file.

```
try {
```

```
        JsonFactory jsonfactory = new JsonFactory();
        JsonParser parser = jsonfactory.createParser(new File("Person.json"));
        ...
        parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

The `nextToken` method returns a `token`. However, the `JsonParser` object keeps track of the current token. In the `while` loop the `nextToken` method returns and advances the parser to the next token. The `getCurrentName` method returns the field name for the `token`. The `while` loop terminates when the last token is reached.

```
while (parser.nextToken() != JsonToken.END_OBJECT) {
    String token = parser.getCurrentName();
    ...
}
```

The body of the loop consists of a series of `if` statements that processes the field by its name. Since the `address` field is an array, another loop will extract each of its elements until the ending array `token` is reached.

```
if ("firstname".equals(token)) {
    parser.nextToken();
    String fname = parser.getText();
    out.println("firstname : " + fname);
}
if ("lastname".equals(token)) {
    parser.nextToken();
    String lname = parser.getText();
    out.println("lastname : " + lname);
}
if ("phone".equals(token)) {
    parser.nextToken();
    long phone = parser.getLongValue();
    out.println("phone : " + phone);
}
if ("address".equals(token)) {
    out.println("address :");
    parser.nextToken();
    while (parser.nextToken() != JsonToken.END_ARRAY) {
        out.println(parser.getText());
    }
}
```

The output of this example follows:

firstname : Smith
lastname : Peter
phone : 8475552222
address :
100 Main Street
Corpus
Oklahoma

However, JSON objects are frequently more complex than the previous example. Here a `Persons.json` file consists of an array of three `persons`:

```
{
    "persons": {
        "groupname": "school",
        "person":
            [
                {"firstname":"Smith",
                 "lastname":"Peter",
                 "phone":8475552222,
                 "address":["100 Main Street","Corpus","Oklahoma"] },
              {"firstname":"King",
                 "lastname":"Sarah",
                 "phone":8475551111,
                 "address":["200 Main Street","Corpus","Oklahoma"] },
              {"firstname":"Frost",
                 "lastname":"Nathan",
                 "phone":8475553333,
                 "address":["300 Main Street","Corpus","Oklahoma"] }
            ]
    }
}
```

To process this file, we use a similar set of code as shown previously. We create the parser and then enter a loop as before:

```
try {
    JsonFactory jsonfactory = new JsonFactory();
    JsonParser parser = jsonfactory.createParser(new File("Person.json"));
    while (parser.nextToken() != JsonToken.END_OBJECT) {
        String token = parser.getCurrentName();
        ...
    }
    parser.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

However, we need to find the `persons` field and then extract each of its elements. The `groupname` field is extracted and displayed as shown here:

```
if ("persons".equals(token)) {
    JsonToken jsonToken = parser.nextToken();
    jsonToken = parser.nextToken();
    token = parser.getCurrentName();
    if ("groupname".equals(token)) {
        parser.nextToken();
        String groupname = parser.getText();
        out.println("Group : " + groupname);
        ...
    }
}
```

Next, we find the `person` field and call a `parsePerson` method to better organize the code:

```
parser.nextToken();
token = parser.getCurrentName();
if ("person".equals(token)) {
    out.println("Found person");
    parsePerson(parser);
}
```

The `parsePerson` method follows which is very similar to the process used in the first example.

```
public void parsePerson(JsonParser parser) throws IOException {
    while (parser.nextToken() != JsonToken.END_ARRAY) {
        String token = parser.getCurrentName();
        if ("firstname".equals(token)) {
            parser.nextToken();
            String fname = parser.getText();
            out.println("firstname : " + fname);
        }
        if ("lastname".equals(token)) {
            parser.nextToken();
            String lname = parser.getText();
            out.println("lastname : " + lname);
        }
        if ("phone".equals(token)) {
            parser.nextToken();
            long phone = parser.getLongValue();
            out.println("phone : " + phone);
        }
        if ("address".equals(token)) {
            out.println("address :");
            parser.nextToken();
            while (parser.nextToken() != JsonToken.END_ARRAY) {
                out.println(parser.getText());
            }
        }
    }
}
```

The output follows:

```
Group : school
Found person
firstname : Smith
lastname : Peter
phone : 8475552222
address :
100 Main Street
Corpus
Oklahoma
firstname : King
lastname : Sarah
phone : 8475551111
address :
200 Main Street
Corpus
```

```
Oklahoma
firstname : Frost
lastname : Nathan
phone : 8475553333address :
300 Main Street
Corpus
Oklahoma
```

Using the JSON tree API

The second approach is to use the tree model. An `ObjectMapper` instance is used to create a `JsonNode` instance using the `Persons.json` file. The `fieldNames` method returns `Iterator` allowing us to process each element of the file.

```
try {
    ObjectMapper mapper = new ObjectMapper();
    JsonNode node = mapper.readTree(new File("Persons.json"));
    Iterator<String> fieldNames = node.fieldNames();
    while (fieldNames.hasNext()) {
        ...
        fieldNames.next();
    }
} catch (IOException ex) {
    // Handle exceptions
}
```

Since the JSON file contains a `persons` field, we will obtain a `JsonNode` instance representing the field and then iterate over each of its elements.

```
JsonNode personsNode = node.get("persons");
Iterator<JsonNode> elements = personsNode.iterator();
while (elements.hasNext()) {
    ...
}
```

Each element is processed one at a time. If the element type is a string, we assume that this is the `groupname` field.

```
JsonNode element = elements.next();
JsonNodeType nodeType = element.getNodeType();

if (nodeType == JsonNodeType.STRING) {
    out.println("Group: " + element.textValue());
}
```

If the element is an array, we assume it contains a series of persons where each person is processed by the `parsePerson` method:

```
if (nodeType == JsonNodeType.ARRAY) {
    Iterator<JsonNode> fields = element.iterator();
    while (fields.hasNext()) {
        parsePerson(fields.next());
    }
}
```

The `parsePerson` method is shown next:

```java
public void parsePerson(JsonNode node) {
    Iterator<JsonNode> fields = node.iterator();
    while(fields.hasNext()) {
        JsonNode subNode = fields.next();
        out.println(subNode.asText());
    }
}
```

The output follows:

```
Group: school
Smith
Peter
8475552222
King
Sarah
8475551111
Frost
Nathan
8475553333
```

There is much more to JSON than we are able to illustrate here. However, this should give you an idea of how this type of data can be handled.

The nitty gritty of cleaning text

Strings are used to support text processing so using a good string library is important. Unfortunately, the `java.lang.String` class has some limitations. To address these limitations, you can either implement your own special string functions as needed or you can use a third-party library.

Creating your own library can be useful, but you will basically be reinventing the wheel. It may be faster to write a simple code sequence to implement some functionality, but to do things right, you will need to test them. Third-party libraries have already been tested and have been used on hundreds of projects. They provide a more efficient way of processing text.

There are several text processing APIs in addition to those found in Java. We will demonstrate two of these:

- **Apache Commons**: https://commons.apache.org/
- **Guava**: https://github.com/google/guava

Java provides many supports for cleaning text data, including methods in the `String` class. These methods are ideal for simple text cleaning and small amounts of data but can also be efficient with larger, complex datasets. We will demonstrate several `String` class methods in a moment. Some of the most helpful `String` class methods are summarized in the following table:

Method Name	Return Type	Description
`trim`	`String`	Removes leading and trailing blank spaces
`toUpperCase/toLowerCase`	`String`	Changes the casing of the entire string
`replaceAll`	`String`	Replaces all occurrences of a character sequence within the string
`contains`	`boolean`	Determines whether a given character sequence exists within the string
`compareTo` `compareToIgnoreCase`	`int`	Compares two strings lexographically and returns an integer representing their relationship
		Determines whether the string matches a given regular

matches	boolean	expression
join	String	Combines two or more strings with a specified delimiter
split	String[]	Separates elements of a given string using a specified delimiter

Many text operations are simplified by the use of regular expressions. Regular expressions use standardized syntax to represent patterns in text, which can be used to locate and manipulate text matching the pattern.

A regular expression is simply a string itself. For example, the string `Hello, my name is Sally` can be used as a regular expression to find those exact words within a given text. This is very specific and not broadly applicable, but we can use a different regular expression to make our code more effective. `Hello, my name is \\w` will match any text that starts with `Hello, my name is` and ends with a word character.

We will use several examples of more complex regular expressions, and some of the more useful syntax options are summarized in the following table. Note each must be double-escaped when used in a Java application.

Option	Description
\d	Any digit: *0-9*
\D	Any non-digit
\s	Any whitespace character
\S	Any non-whitespace character
\w	Any word character (including digits): *A-Z*, *a-z*, and *0-9*
\W	Any non-word character

The size and source of text data varies wildly from application to application but the methods used to transform the data remain the same. You may actually need to read data from a file, but for

simplicity's sake, we will be using a string containing the beginning sentences of Herman Melville's Moby Dick for several examples within this chapter. Unless otherwise specified, the text will assumed to be as shown next:

```
String dirtyText = "Call me Ishmael. Some years ago- never mind how";
dirtyText += " long precisely - having little or no money in my purse,";
dirtyText += " and nothing particular to interest me on shore, I thought";
dirtyText += " I would sail about a little and see the watery part of the
world.";
```

Using Java tokenizers to extract words

Often it is most efficient to analyze text data as tokens. There are multiple tokenizers available in the core Java libraries as well as third-party tokenizers. We will demonstrate various tokenizers throughout this chapter. The ideal tokenizer will depend upon the limitations and requirements of an individual application.

Java core tokenizers

`StringTokenizer` was the first and most basic tokenizer and has been available since Java 1. It is not recommended for use in new development as the `String` class's `split` method is considered more efficient. While it does provide a speed advantage for files with narrowly defined and set delimiters, it is less flexible than other tokenizer options. The following is a simple implementation of the `StringTokenizer` class that splits a string on spaces:

```
StringTokenizer tokenizer = new StringTokenizer(dirtyText," ");
while(tokenizer.hasMoreTokens()){
  out.print(tokenizer.nextToken() + " ");
}
```

When we set the `dirtyText` variable to hold our text from Moby Dick, shown previously, we get the following truncated output:

Call me Ishmael. Some years ago- never mind how long precisely...

`StreamTokenizer` is another core Java tokenizer. `StreamTokenizer` grants more information about the tokens retrieved, and allows the user to specify data types to parse, but is considered more difficult to use than `StreamTokenizer` or the `split` method. The `String` class `split` method is the simplest way to split strings up based on a delimiter, but it does not provide a way to parse the split strings and you can only specify one delimiter for the entire string. For these reasons, it is not a true tokenizer, but it can be useful for data cleaning.

The `Scanner` class is designed to allow you to parse strings into different data types. We used it previously in the *Handling CSV data* section and we will address it again in the *Removing stop words* section.

Third-party tokenizers and libraries

Apache Commons consists of sets of open source Java classes and methods. These provide reusable code that complements the standard Java APIs. One popular class included in the Commons is `StrTokenizer`. This class provides more advanced support than the standard `StringTokenizer` class, specifically more control and flexibility. The following is a simple implementation of the `StrTokenizer`:

```
StrTokenizer tokenizer = new StrTokenizer(text);
while (tokenizer.hasNext()) {
  out.print(tokenizer.next() + " ");
}
```

This operates in a similar fashion to `StringTokenizer` and by default parses tokens on spaces. The constructor can specify the delimiter as well as how to handle double quotes contained in data.

When we use the string from Moby Dick, shown previously, the first tokenizer implementation produces the following truncated output:

Call me Ishmael. Some years ago- never mind how long precisely - having little or no money in my purse...

We can modify our constructor as follows:

```
StrTokenizer tokenizer = new StrTokenizer(text,",");
```

The output for this implementation is:

Call me Ishmael. Some years ago- never mind how long precisely - having
little or no money in my purse
and nothing particular to interest me on shore
I thought I would sail about a little and see the watery part of the world.

Notice how each line is split where commas existed in the original text. This delimiter can be a simple char, as we have shown, or a more complex `StrMatcher` object.

Google Guava is an open source set of utility Java classes and methods. The primary goal of Guava, as with many APIs, is to relieve the burden of writing basic Java utilities so developers can focus on business processes. We are going to talk about two main tools in Guava in this chapter: the `Joiner` class and the `Splitter` class. Tokenization is accomplished in Guava using its `Splitter` class's `split` method. The following is a simple example:

```
Splitter simpleSplit = Splitter.on(',').omitEmptyStrings().trimResults();
Iterable<String> words = simpleSplit.split(dirtyText);
for(String token: words){
  out.print(token);
}
```

This splits each token on commas and produces output like our last example. We can modify the parameter of the `on` method to split on the character of our choosing. Notice the method chaining which allows us to omit empty strings and trim leading and trailing spaces. For these reasons, and other advanced capabilities, Google Guava is considered by some to be the best tokenizer available for Java.

LingPipe is a linguistical toolkit available for language processing in Java. It provides more specialized support for text splitting with its `TokenizerFactory` interface. We implement a LingPipe `IndoEuropeanTokenizerFactory` tokenizer in the *Simple text cleaning* section.

Transforming data into a usable form

Data often needs to be cleaned once it has been acquired. Datasets are often inconsistent, are missing in information, and contain extraneous information. In this section, we will examine some simple ways to transform text data to make it more useful and easier to analyse.

Simple text cleaning

We will use the string shown before from Moby Dick to demonstrate some of the basic `String` class methods. Notice the use of the `toLowerCase` and `trim` methods. Datasets often have non-standard casing and extra leading or trailing spaces. These methods ensure uniformity of our dataset. We also use the `replaceAll` method twice. In the first instance, we use a regular expression to replace all numbers and anything that is not a word or whitespace character with a single space. The second instance replaces all back-to-back whitespace characters with a single space:

```
out.println(dirtyText);
dirtyText =    dirtyText.toLowerCase().replaceAll("[\\d[^\\w\\s]]+", " ");
dirtyText = dirtyText.trim();
while(dirtyText.contains("  ")){
  dirtyText = dirtyText.replaceAll("  ", " ");
}
out.println(dirtyText);
```

When executed, the code produces the following output, truncated:

```
Call me Ishmael. Some years ago- never mind how long precisely -
call me ishmael some years ago never mind how long precisely
```

Our next example produces the same result but approaches the problem with regular expressions. In this case, we replace all of the numbers and other special characters first. Then we use method chaining to standardize our casing, remove leading and trailing spaces, and split our words into a `String` array. The `split` method allows you to break apart text on a given delimiter. In this case, we chose to use the regular expression `\\W`, which represents anything that is not a word character:

```
out.println(dirtyText);
dirtyText = dirtyText.replaceAll("[\\d[^\\w\\s]]+", "");
String[] cleanText = dirtyText.toLowerCase().trim().split("[\\W]+");
for(String clean : cleanText){
  out.print(clean + " ");
}
```

This code produces the same output as shown previously.

Although arrays are useful for many applications, it is often important to recombine text after cleaning. In the next example, we employ the `join` method to combine our words once we have cleaned them. We use the same chained methods as shown previously to clean and split our text. The `join` method joins every word in the array `words` and inserts a space between each word:

```
out.println(dirtyText);
String[] words =   dirtyText.toLowerCase().trim().split("[\\W\\d]+");
String cleanText = String.join(" ", words);
out.println(cleanText);
```

Again, this code produces the same output as shown previously. An alternate version of the `join` method is available using Google Guava. Here is a simple implementation of the same process we used before, but using the Guava `Joiner` class:

```
out.println(dirtyText);
String[] words =   dirtyText.toLowerCase().trim().split("[\\W\\d]+");
String cleanText = Joiner.on(" ").skipNulls().join(words);
out.println(cleanText);
```

This version provides additional options, including skipping nulls, as shown before. The output remains the same.

Removing stop words

Text analysis sometimes requires the omission of common, non-specific words such as *the, and,* or *but*. These words are known as stop words are there are several tools available for removing them from text. There are various ways to store a list of stop words, but for the following examples, we will assume they are contained in a file. To begin, we create a new `Scanner` object to read in our stop words. Then we take the text we wish to transform and store it in an `ArrayList` using the `Arrays` class's `asList` method. We will assume here the text has already been cleaned and normalized. It is essential to consider casing when using `String` class methods —*and* is not the same as *AND* or *And*, although all three may be stop words you wish to eliminate:

```
Scanner readStop = new Scanner(new File("C://stopwords.txt"));
ArrayList<String> words = new    ArrayList<String>
(Arrays.asList((dirtyText));
out.println("Original clean text: " + words.toString());
```

We also create a new `ArrayList` to hold a list of stop words actually found in our text. This will allow us to use the `ArrayList` class `removeAll` method shortly. Next, we use our `Scanner` to read through our file of stop words. Notice how we also call the `toLowerCase` and `trim` methods against each stop word. This is to ensure that our stop words match the formatting in our text. In this example, we employ the `contains` method to determine whether our text contains the given stop word. If so, we add it to our `foundWords` ArrayList. Once we have processed all the stop words, we call `removeAll` to remove them from our text:

```
ArrayList<String> foundWords = new ArrayList();
while(readStop.hasNextLine()){
  String stopWord = readStop.nextLine().toLowerCase();
  if(words.contains(stopWord)){
    foundWords.add(stopWord);
  }
}
words.removeAll(foundWords);
```

```
out.println("Text without stop words: " + words.toString());
```

The output will depend upon the words designated as stop words. If your stop words file contains different words than used in this example, your output will differ slightly. Our output follows:

Original clean text: [call, me, ishmael, some, years, ago, never, mind, how, long, precisely, having, little, or, no, money, in, my, purse, and, nothing, particular, to, interest, me, on, shore, i, thought, i, would, sail, about, a, little, and, see, the, watery, part, of, the, world]
Text without stop words: [call, ishmael, years, ago, never, mind, how, long, precisely

There is also support outside of the standard Java libraries for removing stop words. We are going to look at one example, using LingPipe. In this example, we start by ensuring that our text is normalized in lowercase and trimmed. Then we create a new instance of the `TokenizerFactory` class. We set our factory to use default English stop words and then tokenize the text. Notice that the `tokenizer` method uses a `char` array, so we call `toCharArray` against our text. The second parameter specifies where to begin searching within the text, and the last parameter specifies where to end:

```
text = text.toLowerCase().trim();
TokenizerFactory fact = IndoEuropeanTokenizerFactory.INSTANCE;
fact = new EnglishStopTokenizerFactory(fact);
Tokenizer tok = fact.tokenizer(text.toCharArray(), 0, text.length());
for(String word : tok){
  out.print(word + " ");
}
```

The output follows:

Call me Ishmael. Some years ago- never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world.
call me ishmael . years ago - never mind how long precisely - having little money my purse , nothing particular interest me shore , i thought i sail little see watery part world .

Notice the differences between our previous examples. First of all, we did not clean the text as thoroughly and allowed special characters, such as the hyphen, to remain in the text. Secondly, the LingPipe list of stop words differs from the file we used in the previous example. Some words are removed, but LingPipe was less restrictive and allowed more words to remain in the text. The type and number of stop words you use will depend upon your particular application.

Finding words in text

The standard Java libraries offer support for searching through text for specific tokens. In previous examples, we have demonstrated the `matches` method and regular expressions, which can be useful when searching text. In this example, however, we will demonstrate a simple technique using the `contains` method and the `equals` method to locate a particular string. First, we normalize our text and the word we are searching for to ensure we can find a match. We also create an integer variable to hold the number of times the word is found:

```
dirtyText = dirtyText.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
int count = 0;
```

Next, we call the `contains` method to determine whether the word exists in our text. If it does, we split the text into an array and then loop through, using the `equals` method to compare each word. If we encounter the word, we increment our counter by one. Finally, we display the output to show how many times our word was encountered:

```
if(dirtyText.contains(toFind)){
      String[] words = dirtyText.split(" ");
      for(String word : words){
            if(word.equals(toFind)){
                  count++;
            }
      }
out.println("Found " + toFind + " " + count + " times in the text.");
}
```

In this example, we set `toFind` to the letter `I`. This produced the following output:

Found i 2 times in the text.

We also have the option to use the `Scanner` class to search through an entire file. One helpful method is the `findWithinHorizon` method. This uses a `Scanner` to parse the text up to a given horizon specification. If zero is used for the second parameter, as shown next, the entire `Scanner` will be searched by default:

```
dirtyText = dirtyText.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
Scanner textLine = new Scanner(dirtyText);
out.println("Found " + textLine.findWithinHorizon(toFind, 10));
```

This technique can be more efficient for locating a particular string, but it does make it more difficult to determine where, and how many times, the string was found.

It can also be more efficient to search an entire file using a `BufferedReader`. We specify the file to search and use a try-catch block to catch any IO exceptions. We create a new `BufferedReader` object from our path and process our file as long as the next line is not empty:

```
String path = "C:// MobyDick.txt";
try {
    String textLine = "";
    toFind = toFind.toLowerCase().trim();
    BufferedReader textToClean = new BufferedReader(
        new FileReader(path));
    while((textLine = textToClean.readLine()) != null){
        line++;
        if(textLine.toLowerCase().trim().contains(toFind)){
            out.println("Found " + toFind + " in " + textLine);
          }
    }
    textToClean.close();
} catch (IOException ex) {
    // Handle exceptions
}
```

We again test our data by searching for the word I in the first sentences of Moby Dick. The truncated output follows:

Found i in Call me Ishmael...

Finding and replacing text

We often not only want to find text but also replace it with something else. We begin our next example much like we did the previous examples, by specifying our text, our text to locate, and invoking the `contains` method. If we find the text, we call the `replaceAll` method to modify our string:

```
text = text.toLowerCase().trim();
toFind = toFind.toLowerCase().trim();
out.println(text);

if(text.contains(toFind)){
    text = text.replaceAll(toFind, replaceWith);
    out.println(text);
}
```

To test this code, we set `toFind` to the word I and `replaceWith` to Ishmael. Our output follows:

**call me ishmael. some years ago- never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, i thought i would sail about a little and see the watery part of the world.
call me ishmael. some years ago- never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, Ishmael thought Ishmael would sail about a little and see the watery part of the world.**

Apache Commons also provides a `replace` method with several variations in the `StringUtils` class. This class provides much of the same functionality as the `String` class, but with more flexibility and options. In the following example, we use our string from Moby Dick and replace

all instances of the word `me` with `X` to demonstrate the `replace` method:

```
out.println(text);
out.println(StringUtils.replace(text, "me", "X"));
```

The truncated output follows:

Call me Ishmael. Some years ago- never mind how long precisely -
Call X Ishmael. SoX years ago- never mind how long precisely -

Notice how every instance of `me` has been replaced, even those instances contained within other words, such as `some`. This can be avoided by adding spaces around `me`, although this will ignore any instances where me is at the end of the sentence, like me. We will examine a better alternative using Google Guava in a moment.

The `StringUtils` class also provides a `replacePattern` method that allows you to search for and replace text based upon a regular expression. In the following example, we replace all non-word characters, such as hyphens and commas, with a single space:

```
out.println(text);
text = StringUtils.replacePattern(text, "\\W\\s", " ");
out.println(text);
```

This will produce the following truncated output:

Call me Ishmael. Some years ago- never mind how long precisely -
Call me Ishmael Some years ago never mind how long precisely

Google Guava provides additional support for matching and modify text data using the `CharMatcher` class. `CharMatcher` not only allows you to find data matching a particular char pattern, but also provides options as to how to handle the data. This includes allowing you to retain the data, replace the data, and trim whitespaces from within a particular string.

In this example, we are going to use the `replace` method to simply replace all instances of the word `me` with a single space. This will produce series of empty spaces within our text. We will then collapse the extra whitespace using the `trimAndCollapseFrom` method and print our string again:

```
text = text.replace("me", " ");
out.println("With double spaces: " + text);
String spaced = CharMatcher.WHITESPACE.trimAndCollapseFrom(text, ' ');
out.println("With double spaces removed: " + spaced);
```

Our output is truncated as follows:

With double spaces: Call Ishmael. So years ago- ...
With double spaces removed: Call Ishmael. So years ago- ...

Data imputation

Data imputation refers to the process of identifying and replacing missing data in a given dataset. In almost any substantial case of data analysis, missing data will be an issue, and it needs to be addressed before data can be properly analysed. Trying to process data that is missing information is a lot like trying to understand a conversation where every once in while a word is dropped. Sometimes we can understand what is intended. In other situations, we may be completely lost as to what is trying to be conveyed.

Among statistical analysts, there exist differences of opinion as to how missing data should be handled but the most common approaches involve replacing missing data with a reasonable estimate or with an empty or null value.

To prevent skewing and misalignment of data, many statisticians advocate for replacing missing data with values representative of the average or expected value for that dataset. The methodology for determining a representative value and assigning it to a location within the data will vary depending upon the data and we cannot illustrate every example in this chapter. However, for example, if a dataset contained a list of temperatures across a range of dates, and one date was missing a temperature, that date can be assigned a temperature that was the average of the temperatures within the dataset.

We will examine a rather trivial example to demonstrate the issues surrounding data imputation. Let's assume the variable `tempList` contains average temperature data for each month of one year. Then we perform a simple calculation of the average and print out our results:

```
double[] tempList = {50,56,65,70,74,80,82,90,83,78,64,52};
double sum = 0;
for(double d : tempList){
      sum += d;
}
out.printf("The average temperature is %1$,.2f", sum/12);
```

Notice that for the numbers used in this execution, the output is as follows:

The average temperature is 70.33

Next we will mimic missing data by changing the first element of our array to zero before we calculate our `sum`:

```
double sum = 0;
tempList[0] = 0;
for(double d : tempList){
      sum += d;
}
out.printf("The average temperature is %1$,.2f", sum/12);
```

This will change the average temperature displayed in our output:

```
The average temperature is 66.17
```

Notice that while this change may seem rather minor, it is statistically significant. Depending upon the variation within a given dataset and how far the average is from zero or some other substituted value, the results of a statistical analysis may be significantly skewed. This does not mean zero should never be used as a substitute for null or otherwise invalid values, but other alternatives should be considered.

One alternative approach can be to calculate the average of the values in the array, excluding zeros or nulls, and then substitute the average in each position with missing data. It is important to consider the type of data and purpose of data analysis when making these decisions. For example, in the preceding example, will zero always be an invalid average temperature? Perhaps not if the temperatures were averages for Antarctica.

When it is essential to handle null data, Java's `Optional` class provides helpful solutions. Consider the following example, where we have a list of names stored as an array. We have set one value to `null` for the purposes of demonstrating these methods:

```
String useName = "";
String[] nameList =
        {"Amy","Bob","Sally","Sue","Don","Rick",null,"Betsy"};
Optional<String> tempName;
for(String name : nameList){
        tempName = Optional.ofNullable(name);
        useName = tempName.orElse("DEFAULT");
        out.println("Name to use = " + useName);
}
```

We first created a variable called `useName` to hold the name we will actually print out. We also created an instance of the `Optional` class called `tempName`. We will use this to test whether a value in the array is null or not. We then loop through our array and create and call the `Optional` class `ofNullable` method. This method tests whether a particular value is null or not. On the next line, we call the `orElse` method to either assign a value from the array to `useName` or, if the element is null, assign `DEFAULT`. Our output follows:

```
Name to use = Amy
Name to use = Bob
Name to use = Sally
Name to use = Sue
Name to use = Don
Name to use = Rick
Name to use = DEFAULT
Name to use = Betsy
```

The `Optional` class contains several other methods useful for handling potential null data. Although there are other ways to handle such instances, this Java 8 addition provides simpler and more elegant solutions to a common data analysis problem.

Subsetting data

It is not always practical or desirable to work with an entire set of data. In these cases, we may want to retrieve a subset of data to either work with or remove entirely from the dataset. There are a few ways of doing this supported by the standard Java libraries. First, we will use the `subSet` method of the `SortedSet` interface. We will begin by storing a list of numbers in a `TreeSet`. We then create a new `TreeSet` object to hold the subset retrieved from the list. Next, we print out our original list:

```
Integer[] nums = {12, 46, 52, 34, 87, 123, 14, 44};
TreeSet<Integer> fullNumsList = new TreeSet<Integer>(new
ArrayList<>(Arrays.asList(nums)));
SortedSet<Integer> partNumsList;
out.println("Original List: " + fullNumsList.toString()
    + " " + fullNumsList.last());
```

The `subSet` method takes two parameters, which specify the range of integers within the data we want to retrieve. The first parameter is included in the results while the second is exclusive. In our example that follows, we want to retrieve a subset of all numbers between the first number in our array `12` and `46`:

```
partNumsList = fullNumsList.subSet(fullNumsList.first(), 46);
out.println("SubSet of List: " + partNumsList.toString()
    + " " + partNumsList.size());
```

Our output follows:

```
Original List: [12, 14, 34, 44, 46, 52, 87, 123]
SubSet of List: [12, 14, 34, 44]
```

Another option is to use the `stream` method in conjunction with the `skip` method. The `stream` method returns a Java 8 Stream instance which iterates over the set. We will use the same `numsList` as in the previous example, but this time we will specify how many elements to skip with the `skip` method. We will also use the `collect` method to create a new `Set` to hold the new elements:

```
out.println("Original List: " + numsList.toString());
Set<Integer> fullNumsList = new TreeSet<Integer>(numsList);
Set<Integer> partNumsList = fullNumsList
        .stream()
        .skip(5)
        .collect(toCollection(TreeSet::new));
out.println("SubSet of List: " + partNumsList.toString());
```

When we print out the new subset, we get the following output where the first five elements of the sorted set are skipped. Because it is a `SortedSet`, we will actually be omitting the five lowest numbers:

```
Original List: [12, 46, 52, 34, 87, 123, 14, 44]
```

SubSet of List: [52, 87, 123]

At times, data will begin with blank lines or header lines that we wish to remove from our dataset to be analysed. In our final example, we will read data from a file and remove all blank lines. We use a `BufferedReader` to read our data and employ a lambda expression to test for a blank line. If the line is not blank, we print the line to the screen:

```java
try (BufferedReader br = new BufferedReader(new FileReader("C:\\text.txt")))
{
   br
        .lines()
        .filter(s -> !s.equals(""))
        .forEach(s -> out.println(s));
} catch (IOException ex) {
   // Handle exceptions
}
```

Sorting text

Sometimes it is necessary to sort data during the cleaning process. The standard Java library provides several resources for accomplishing different types of sorts, with improvements added with the release of Java 8. In our first example, we will use the `Comparator` interface in conjunction with a lambda expression.

We start by declaring our `Comparator` variable `compareInts`. The first set of parenthesis after the equals sign contains the parameters to be passed to our method. Within the lambda expression, we call the `compare` method, which determines which integer is larger:

```
Comparator<Integer> compareInts = (Integer first, Integer second) ->
    Integer.compare(first, second);
```

We can now call the `sort` method as we did previously:

```
Collections.sort(numsList,compareInts);
out.println("Sorted integers using Lambda: " + numsList.toString());
```

Our output follows:

Sorted integers using Lambda: [12, 14, 34, 44, 46, 52, 87, 123]

We then mimic the process with our `wordsList`. Notice the use of the `compareTo` method rather than `compare`:

```
Comparator<String> compareWords = (String first, String second) ->
first.compareTo(second);
Collections.sort(wordsList,compareWords);
out.println("Sorted words using Lambda: " + wordsList.toString());
```

When this code is executed, we should see the following output:

Sorted words using Lambda: [boat, cat, dog, house, road, zoo]

In our next example, we are going to use the `Collections` class to perform basic sorting on `String` and integer data. For this example, `wordList` and `numsList` are both `ArrayList` and are initialized as follows:

```
List<String> wordsList
        = Stream.of("cat", "dog", "house", "boat", "road", "zoo")
        .collect(Collectors.toList());
List<Integer> numsList = Stream.of(12, 46, 52, 34, 87, 123, 14, 44)
        .collect(Collectors.toList());
```

First, we will print our original version of each list followed by a call to the `sort` method. We then display our data, sorted in ascending fashion:

```
out.println("Original Word List: " + wordsList.toString());
Collections.sort(wordsList);
out.println("Ascending Word List: " + wordsList.toString());
out.println("Original Integer List: " + numsList.toString());
Collections.sort(numsList);
out.println("Ascending Integer List: " + numsList.toString());
```

The output follows:

Original Word List: [cat, dog, house, boat, road, zoo]
Ascending Word List: [boat, cat, dog, house, road, zoo]
Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Ascending Integer List: [12, 14, 34, 44, 46, 52, 87, 123]

Next, we will replace the sort method with the reverse method of the Collections class in our integer data example. This method simply takes the elements and stores them in reverse order:

```
 out.println("Original Integer List: " + numsList.toString());
 Collections.reverse(numsList);
 out.println("Reversed Integer List: " + numsList.toString());
```

The output displays our new numsList:

Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Reversed Integer List: [44, 14, 123, 87, 34, 52, 46, 12]

In our next example, we handle the sort using the Comparator interface. We will continue to use our numsList and assume that no sorting has occurred yet. First we create two objects that implement the Comparator interface. The sort method will use these objects to determine the desired order when comparing two elements. The expression Integer::compare is a Java 8 method reference. This is can be used where a lambda expression is used:

```
out.println("Original Integer List: " + numsList.toString());
Comparator<Integer> basicOrder = Integer::compare;
Comparator<Integer> descendOrder = basicOrder.reversed();
Collections.sort(numsList,descendOrder);
out.println("Descending Integer List: " + numsList.toString());
```

After we execute this code, we will see the following output:

Original Integer List: [12, 46, 52, 34, 87, 123, 14, 44]
Descending Integer List: [123, 87, 52, 46, 44, 34, 14, 12]

In our last example, we will attempt a more complex sort involving two comparisons. Let's assume there is a Dog class that contains two properties, name and age, along with the necessary accessor methods. We will begin by adding elements to a new ArrayList and then printing the names and ages of each Dog:

```
ArrayList<Dogs> dogs = new ArrayList<Dogs>();
```

```
dogs.add(new Dogs("Zoey", 8));
dogs.add(new Dogs("Roxie", 10));
dogs.add(new Dogs("Kylie", 7));
dogs.add(new Dogs("Shorty", 14));
dogs.add(new Dogs("Ginger", 7));
dogs.add(new Dogs("Penny", 7));
out.println("Name " + " Age");
for(Dogs d : dogs){
     out.println(d.getName() + " " + d.getAge());
}
```

Our output should resemble:

```
Name Age
Zoey 8
Roxie 10
Kylie 7
Shorty 14
Ginger 7
Penny 7
```

Next, we are going to use method chaining and the double colon operator to reference methods from the Dog class. We first call comparing followed by thenComparing to specify the order in which comparisons should occur. When we execute the code, we expect to see the Dog objects sorted first by Name and then by Age:

```
dogs.sort(Comparator.comparing(Dogs::getName).thenComparing(Dogs::getAge));
out.println("Name " + " Age");
for(Dogs d : dogs){
     out.println(d.getName() + " " + d.getAge());
}
```

Our output follows:

```
Name Age
Ginger 7
Kylie 7
Penny 7
Roxie 10
Shorty 14
Zoey 8
```

Now we will switch the order of comparison. Notice how the age of the dog takes priority over the name in this version:

```
dogs.sort(Comparator.comparing(Dogs::getAge).thenComparing(Dogs::getName));
out.println("Name " + " Age");
for(Dogs d : dogs){
     out.println(d.getName() + " " + d.getAge());
}
```

And our output is:

```
Name Age
Ginger 7
Kylie 7
Penny 7
Zoey 8
Roxie 10
Shorty 14
```

Data validation

Data validation is an important part of data science. Before we can analyze and manipulate data, we need to verify that the data is of the type expected. We have organized our code into simple methods designed to accomplish very basic validation tasks. The code within these methods can be adapted into existing applications.

Validating data types

Sometimes we simply need to validate whether a piece of data is of a specific type, such as integer or floating point data. We will demonstrate in the next example how to validate integer data using the `validateInt` method. This technique is easily modified for the other major data types supported in the standard Java library, including `Float` and `Double`.

We need to use a try-catch block here to catch a `NumberFormatException`. If an exception is thrown, we know our data is not a valid integer. We first pass our text to be tested to the `parseInt` method of the `Integer` class. If the text can be parsed as an integer, we simply print out the integer. If an exception is thrown, we display information to that effect:

```
public static void validateInt(String toValidate){
try{
    int validInt = Integer.parseInt(toValidate);
    out.println(validInt + " is a valid integer");
}catch(NumberFormatException e){
    out.println(toValidate + " is not a valid integer");

}
```

We will use the following method calls to test our method:

```
validateInt("1234");
validateInt("Ishmael");
```

The output follows:

```
1234 is a valid integer
Ishmael is not a valid integer
```

The Apache Commons contain an `IntegerValidator` class with additional useful functionalities. In this first example, we simply duplicate the process from before, but use `IntegerValidator` methods to accomplish our goal:

```
public static String validateInt(String text){
    IntegerValidator intValidator = IntegerValidator.getInstance();
    if(intValidator.isValid(text)){
        return text + " is a valid integer";
    }else{
        return text + " is not a valid integer";
    }
}
```

We again use the following method calls to test our method:

```
validateInt("1234");
validateInt("Ishmael");
```

The output follows:

1234 is a valid integer
Ishmael is not a valid integer

The `IntegerValidator` class also provides methods to determine whether an integer is greater than or less than a specific value, compare the number to a ranger of numbers, and convert `Number` objects to `Integer` objects. Apache Commons has a number of other validator classes. We will examine a few more in the rest of this section.

Validating dates

Many times our data validation is more complex than simply determining whether a piece of data is the correct type. When we want to verify that the data is a date for example, it is insufficient to simply verify that it is made up of integers. We may need to include hyphens and slashes, or ensure that the year is in two-digit or four-digit format.

To do this, we have created another simple method called `validateDate`. The method takes two `String` parameters, one to hold the date to validate and the other to hold the acceptable date format. We create an instance of the `SimpleDateFormat` class using the format specified in the parameter. Then we call the `parse` method to convert our `String` date to a `Date` object. Just as in our previous integer example, if the data cannot be parsed as a date, an exception is thrown and the method returns. If, however, the `String` can be parsed to a date, we simply compare the format of the test date with our acceptable format to determine whether the date is valid:

```
public static String validateDate(String theDate, String dateFormat){
     try {
          SimpleDateFormat format = new SimpleDateFormat(dateFormat);
          Date test = format.parse(theDate);
          if(format.format(test).equals(theDate)){
               return theDate.toString() + " is a valid date";
          }else{
               return theDate.toString() + " is not a valid date";
          }
     } catch (ParseException e) {
          return theDate.toString() + " is not a valid date";
     }
}
```

We make the following method calls to test our method:

```
String dateFormat = "MM/dd/yyyy";
out.println(validateDate("12/12/1982",dateFormat));
out.println(validateDate("12/12/82",dateFormat));
out.println(validateDate("Ishmael",dateFormat));
```

The output follows:

```
12/12/1982 is a valid date
12/12/82 is not a valid date
Ishmael is not a valid date
```

This example highlights why it is important to consider the restrictions you place on data. Our second method call did contain a legitimate date, but it was not in the format we specified. This is good if we are looking for very specifically formatted data. But we also run the risk of leaving out useful data if we are too restrictive in our validation.

Validating e-mail addresses

It is also common to need to validate e-mail addresses. While most e-mail addresses have the @ symbol and require at least one period after the symbol, there are many variations. Consider that each of the following examples can be valid e-mail addresses:

- myemail@mail.com
- MyEmail@some.mail.com
- My.Email.123!@mail.net

One option is to use regular expressions to attempt to capture all allowable e-mail addresses. Notice that the regular expression used in the method that follows is very long and complex. This can make it easy to make mistakes, miss valid e-mail addresses, or accept invalid addresses as valid. But a carefully crafted regular expression can be a very powerful tool.

We use the `Pattern` and `Matcher` classes to compile and execute our regular expression. If the pattern of the e-mail we pass in matches the regular expression we defined, we will consider that text to be a valid e-mail address:

```
public static String validateEmail(String email) {
    String emailRegex = "^[a-zA-Z0-9.!$'*+/=?^_`{|}~-" +
        "]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\." +
        "[0-9]{1,3}\\])|(([a-zAZ\\-0-9]+\\.)+[a-zA-Z]{2,}))$";
    Pattern.compile(emailRegex);
    Matcher matcher = pattern.matcher(email);
    if(matcher.matches()){
        return email + " is a valid email address";
    }else{
        return email + " is not a valid email address";
    }
}
```

We make the following method calls to test our data:

```
out.println(validateEmail("myemail@mail.com"));
out.println(validateEmail("My.Email.123!@mail.net"));
out.println(validateEmail("myEmail"));
```

The output follows:

myemail@mail.com is a valid email address
My.Email.123!@mail.net is a valid email address
myEmail is not a valid email address

There is a standard Java library for validating e-mail addresses as well. In this example, we use the `InternetAddress` class to validate whether a given string is a valid e-mail address or not:

```java
public static String validateEmailStandard(String email){
    try{
        InternetAddress testEmail = new InternetAddress(email);
        testEmail.validate();
        return email + " is a valid email address";
    }catch(AddressException e){
        return email + " is not a valid email address";
    }
}
```

When tested against the same data as in the previous example, our output is identical. However, consider the following method call:

```java
out.println(validateEmailStandard("myEmail@mail"));
```

Despite not being in standard e-mail format, the output is as follows:

myEmail@mail is a valid email address

Additionally, the `validate` method by default accepts local e-mail addresses as valid. This is not always desirable, depending upon the purpose of the data.

One last option we will look at is the Apache Commons `EmailValidator` class. This class's `isValid` method examines an e-mail address and determines whether it is valid or not. Our `validateEmail` method shown previously is modified as follows to use `EmailValidator`:

```java
public static String validateEmailApache(String email){
    email = email.trim();
    EmailValidator eValidator = EmailValidator.getInstance();
    if(eValidator.isValid(email)){
        return email + " is a valid email address.";
    }else{
        return email + " is not a valid email address.";
    }
}
```

Validating ZIP codes

Postal codes are generally formatted specific to their country or local requirements. For this reason, regular expressions are useful because they can accommodate any postal code required. The example that follows demonstrates how to validate a standard United States postal code, with or without the hyphen and last four digits:

```
public static void validateZip(String zip){
    String zipRegex = "^[0-9]{5}(?:-[0-9]{4})?$";
    Pattern pattern = Pattern.compile(zipRegex);
    Matcher matcher = pattern.matcher(zip);
    if(matcher.matches()){
        out.println(zip + " is a valid zip code");
    }else{
        out.println(zip + " is not a valid zip code");
    }
}
```

We make the following method calls to test our data:

```
out.println(validateZip("12345"));
out.println(validateZip("12345-6789"));
out.println(validateZip("123"));
```

The output follows:

12345 is a valid zip code
12345-6789 is a valid zip code
123 is not a valid zip code

Validating names

Names can be especially tricky to validate because there are so many variations. There are no industry standards or technical limitations, other than what characters are available on the keyboard. For this example, we have chosen to use Unicode in our regular expression because it allows us to match any character from any language. The Unicode property \\p{L} provides this flexibility. We also use \\s-', to allow spaces, apostrophes, commas, and hyphens in our name fields. It is possible to perform string cleaning, as discussed earlier in this chapter, before attempting to match names. This will simplify the regular expression required:

```
public static void validateName(String name){
    String nameRegex = "^[\\p{L}\\s-',]+$";
    Pattern pattern = Pattern.compile(nameRegex);
    Matcher matcher = pattern.matcher(name);
    if(matcher.matches()){
        out.println(name + " is a valid name");
    }else{
        out.println(name + " is not a valid name");
    }
}
```

We make the following method calls to test our data:

```
validateName("Bobby Smith, Jr.");
validateName("Bobby Smith the 4th");
validateName("Albrecht Müller");
validateName("François Moreau");
```

The output follows:

Bobby Smith, Jr. is a valid name
Bobby Smith the 4th is not a valid name
Albrecht Müller is a valid name
François Moreau is a valid name

Notice that the comma and period in `Bobby Smith, Jr.` are acceptable, but the `4` in `4th` is not. Additionally, the special characters in `François` and `Müller` are considered valid.

Cleaning images

While image processing is a complex task, we will introduce a few techniques to clean and extract information from an image. This will provide the reader with some insight into image processing. We will also demonstrate how to extract text data from an image using **Optical Character Recognition (OCR).**

There are several techniques used to improve the quality of an image. Many of these require tweaking of parameters to get the improvement desired. We will demonstrate how to:

- Enhance an image's contrast
- Smooth an image
- Brighten an image
- Resize an image
- Convert images to different formats

We will use OpenCV (http://opencv.org/), an open source project for image processing. There are several classes that we will use:

- `Mat`: This represents an n-dimensional array holding image data such as channel, grayscale, or color values
- `Imgproc`: Possesses many methods that process an image
- `Imgcodecs`: Possesses methods to read and write image files

The OpenCV Javadocs is found at http://docs.opencv.org/java/2.4.9/. In the examples that follow, we will use Wikipedia images since they can be freely downloaded. Specifically we will use the following images:

- **Parrot image**: https://en.wikipedia.org/wiki/Grayscale#/media/File:Grayscale_8bits_palette_sampl
- **Cat image**: https://en.wikipedia.org/wiki/Cat#/media/File:Kittyply_edit1.jpg

Changing the contrast of an image

Here we will demonstrate how to enhance a black-and-white image of a parrot. The `Imgcodecs` class's `imread` method reads in the image. Its second parameter specifies the type of color used by the image, which is grayscale in this case. A new `Mat` object is created for the enhanced image using the same size and color type as the original.

The actual work is performed by the `equalizeHist` method. This equalizes the histogram of the image which has the effect of normalizing the brightness and increases the contrast of the image. An image histogram is a histogram representing the tonal distribution of an image. **Tonal** is also referred to as lightness. It represents the variation in the brightness found in an image.

The last step is to write out the image.

```
Mat source = Imgcodecs.imread("GrayScaleParrot.png",
        Imgcodecs.CV_LOAD_IMAGE_GRAYSCALE);
Mat destination = new Mat(source.rows(), source.cols(), source.type());
Imgproc.equalizeHist(source, destination);
Imgcodecs.imwrite("enhancedParrot.jpg", destination);
```

The following is the original image:

The enhanced image follows:

Smoothing an image

Smoothing an image, also called **blurring**, will make the edges of an image smoother. Blurring is the process of making an image less distinct. We recognize blurred objects when we take a picture with the camera out of focus. Blurring can be used for special effects. Here, we will use it to create an image that we will then sharpen.

The following example loads an image of a cat and repeatedly applies the `blur` method to the image. In this example, the process is repeated `25` times. Increasing the number of iterations will result in more blur or smoothing.

The third argument of the blur method is the blurring kernel size. The kernel is a matrix of pixels, 3 by 3 in this example, that is used for convolution. This is the process of multiplying each element of an image by weighted values of its neighbors. This allows neighboring values to effect an element's value:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat destination = source.clone();
for (int i = 0; i < 25; i++) {
    Mat sourceImage = destination.clone();
    Imgproc.blur(sourceImage, destination, new Size(3.0, 3.0));
}
Imgcodecs.imwrite("smoothCat.jpg", destination);
```

The following is the original image:

The enhanced image follows:

Brightening an image

The `convertTo` method provides a means of brightening an image. The original image is copied to a new image where the contrast and brightness is adjusted. The first parameter is the destination image. The second specifies that the type of image should not be changed. The third and fourth parameters control the contrast and brightness respectively. The first value is multiplied by this value while the second is added to the multiplied value:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat destination = new Mat(source.rows(), source.cols(),
        source.type());
source.convertTo(destination, -1, 1, 50);
Imgcodecs.imwrite("brighterCat.jpg", destination);
```

The enhanced image follows:

Resizing an image

Sometimes it is desirable to resize an image. The `resize` method shown next illustrates how this is done. The image is read in and a new `Mat` object is created. The `resize` method is then applied where the width and height are specified in the `Size` object parameter. The resized image is then saved:

```
Mat source = Imgcodecs.imread("cat.jpg");
Mat resizeimage = new Mat();
Imgproc.resize(source, resizeimage, new Size(250, 250));
Imgcodecs.imwrite("resizedCat.jpg", resizeimage);
```

The enhanced image follows:

Converting images to different formats

Another common operation is to convert an image that uses one format into an image that uses a different format. In OpenCV, this is easy to accomplish as shown next. The image is read in and then immediately written out. The extension of the file is used by the `imwrite` method to convert the image to the new format:

```
Mat source = Imgcodecs.imread("cat.jpg");
Imgcodecs.imwrite("convertedCat.jpg", source);
Imgcodecs.imwrite("convertedCat.jpeg", source);
Imgcodecs.imwrite("convertedCat.webp", source);
Imgcodecs.imwrite("convertedCat.png", source);
Imgcodecs.imwrite("convertedCat.tiff", source);
```

The images can now be used for specialized processing if necessary.

Summary

Many times, half the battle in data science is manipulating data so that it is clean enough to work with. In this chapter, we examined many techniques for taking real-world, messy data and transforming it into workable datasets. This process is generally known as data cleaning, wrangling, reshaping, or munging. Our focus was on core Java techniques, but we also examined third-party libraries.

Before we can clean data, we need to have a solid understanding of the format of our data. We discussed CSV data, spreadsheets, PDF, and JSON file types, as well as provided several examples of manipulating text file data. As we examined text data, we looked at multiple approaches for processing the data, including tokenizers, `Scanners`, and `BufferedReaders`. We showed ways to perform simple cleaning operations, remove stop words, and perform find and replace functions.

This chapter also included a discussion on data imputation and the importance of identifying and rectifying missing data situations. Missing data can cause problems during data analysis and we proposed different methods for dealing with this problem. We demonstrated how to retrieve subsets of data and sort data as well.

Finally, we discussed image cleaning and demonstrated several methods of modifying image data. This included changing contrast, smoothing, brightening, and resizing information. We concluded with a discussion on extracting text imposed on an image.

With this background, we will introduce basic statistical methods and their Java support in the next chapter.

Chapter 4. Data Visualization

The human mind is often good at seeing patterns, trends, and outliers in visual representations. The large amount of data present in many data science problems can be analyzed using visualization techniques. Visualization is appropriate for a wide range of audiences, ranging from analysts, to upper-level management, to clientele. In this chapter, we present various visualization techniques and demonstrate how they are supported in Java.

In this chapter, we will illustrate how to create different types of graph, plot, and chart. The majority of the examples use JavaFX, with a few using a free library called **GRAphing Library** (**GRAL**). There are several open source Java plotting libraries available. A brief comparison of several of these libraries can be found at https://github.com/eseifert/gral/wiki/comparison. We chose JavaFX because it is packaged as part of Java SE.

GRAL is used to illustrate plots that are not as easily created using JavaFX. GRAL is a free Java library useful for creating a variety of charts and graphs. This graphing library provides flexibility in types of plots, axis formatting, and export options. GRAL resources (http://trac.erichseifert.de/gral/) include example code and helpful how to sections.

Visualization is an important step in data analysis because it allows us to conceive of large datasets in practical and meaningful ways. We can look at small datasets of values and perhaps draw conclusions from the patterns we see, but this is an overwhelming and unreliable process. Using visualization tools helps us identify potential problems or unexpected data results, as well as construct meaningful interpretations of good data.

One example of the usefulness of data visualization comes with the presence of **outliers**. Visualizing data allows us to quickly see data results significantly outside of our expectations, and we can choose how to modify the data to build a clean and usable dataset. This process allows us to see errors quickly and deal with them before they become a problem later on. Additionally, visualization allows us to easily classify information and help analysts organize their inquiries in a manner best suited to their particular dataset.

Understanding plots and graphs

There are many types of visual expression available to aid in visualization. We are going to briefly discuss the most common and useful ones, and then demonstrate several Java techniques for achieving these types of expression. The choice of graph, or other visualization tool will depend upon the dataset and application needs and constraints.

A **bar chart** is a very common technique for displaying relationships in data. In this type of graph, data is represented in either vertical or horizontal bars placed along an *X* and Y axis. The data is scaled so the values represented by each bar can be compared to one another. The following is a simple example of a bar chart we will create in the *Using country as the category* section:
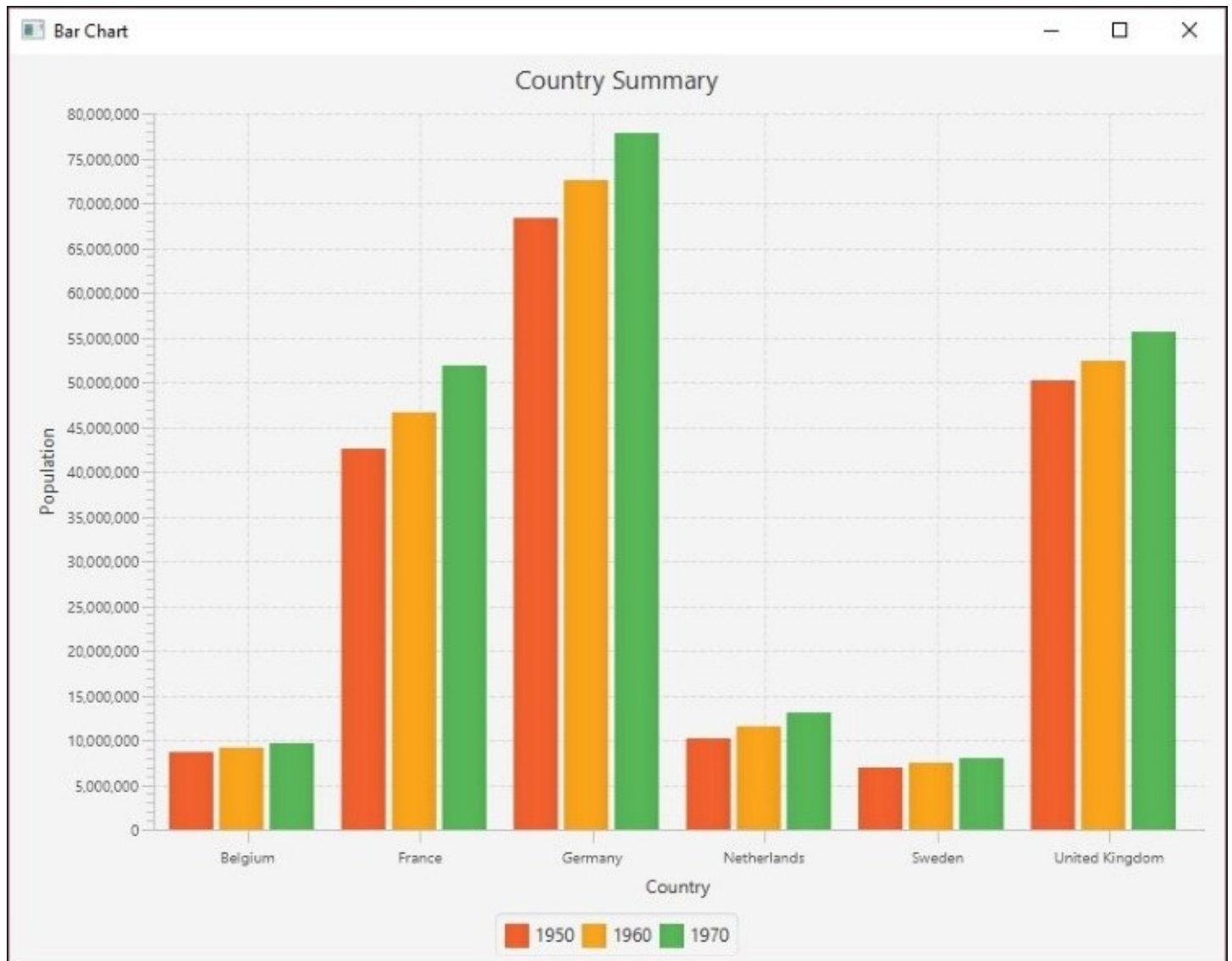
A **pie chart** is most useful when you want to demonstrate a value in relation to a larger set. Think of this as a way to visualize how large the piece of pie is in relation to the entire pie. The following is a simple example of a pie chart showing the distribution of population for selected

European countries:
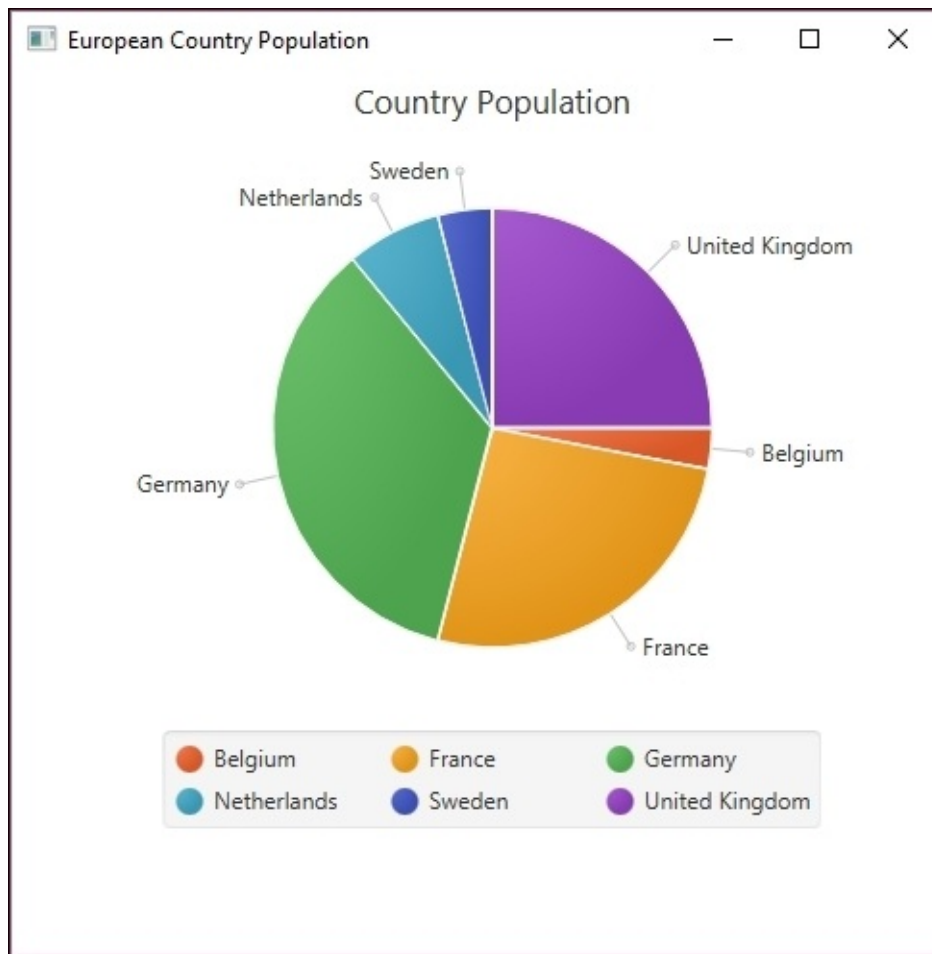
Time series graphs are a special type of graph used for displaying time-related values. These are most appropriate when the data analysis requires an understanding of how data changes over a period of time. In these graphs, the vertical axis corresponds to the values and the horizontal axis corresponds to particular points in time. In particular, this type of graph can be useful for identifying trends across time,or suggesting correlations between data values and particular events within a given time period.

For example, stock prices and home prices will change, but their rate of change varies. Pollution levels and crime rates also change over time. There are several techniques that visualize this type of data. Often, specific values are not as important as their trend over time.

An **index chart** is also called a line chart. **Line charts** use the *X* and *Y* axis to plot points on a grid. They can be used to represent time series data. These points are connected by lines, and these lines are used to compare values of multiple data at one time. This comparison is usually achieved by plotting independent variables, such as time, along the *X* axis, and independent variables, such as frequency or percentages, along the *Y* axis.

The following is a simple example of an index chart showing the distribution of population for

selected European countries:
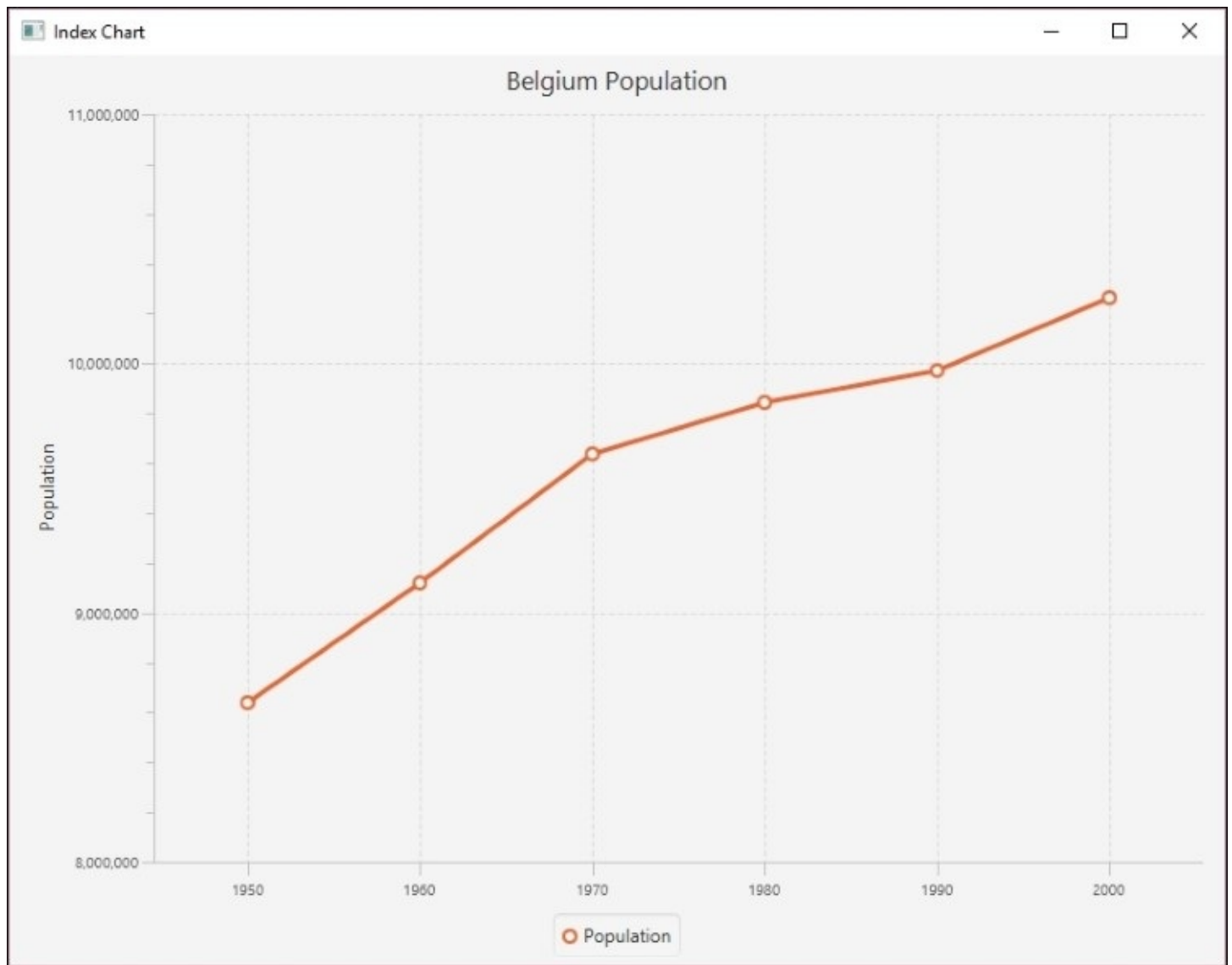
When we wish to arrange larger amounts of data in a compact and useful manner, we may opt for a stem and leaf plot. This type of visual expression allows you to demonstrate the correlation of one value to many values in a readable manner. The stem refers to a data value, and the leaves are the corresponding data points. One common example of this is a train timetable. In the following table, the departure times for a train are listed:

06:15	06 :20	06:25	06:30
06:40	06:45	06:55	07:15
07:20	07:25	07:30	07:40

07:45	07:55	08:00	08:12
08:24	08:36	08:48	09:00
09:12	09:24	09:36	09:48
10:00	10:12	10:24	10:36
10:48			

However, this table can be hard to read. Instead, in the following partial stem and leaf plot, the stem represents the hours at which a train may depart, while the leaves represent the minutes within each hour:

Hour	Minute
06	:15 :20 :25 :30 :40 :45 :55
07	:15 :20 :25 :30 :40 :45 :55
08	:00 :12 :24 :36 :48
09	:00 :12 :24 :36 :48
10	:00 :12 :24 :36 :48

This is much easier to read and process.

A very popular form of visualization in statistical analysis is the **histogram**. Histograms allow you to display frequencies within data using bars, similar to a bar chart. The main difference is that histograms are used to identify frequencies and trends within a dataset while bar charts are used to compare specific data values within a dataset. The following is an example of a histogram we will create in the *Creating histograms* section:
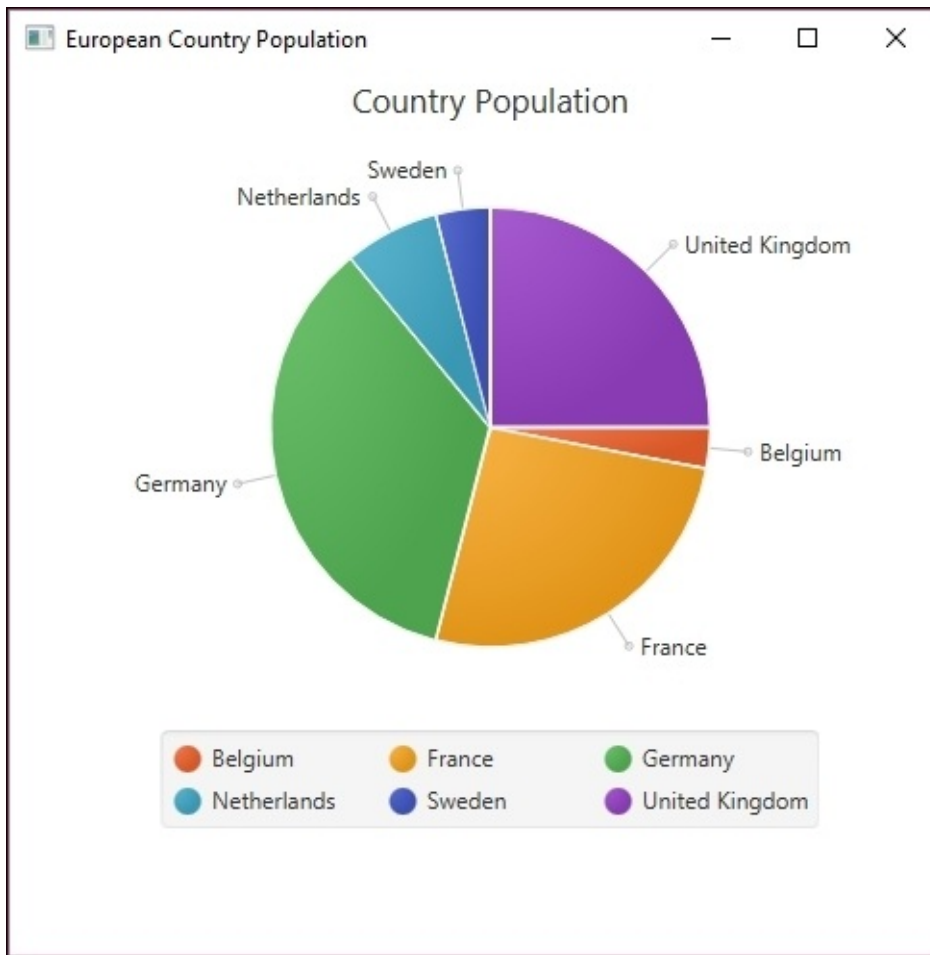
A **scatter plot** is simply collections of points, and analysis techniques, such as correlation or regression, can be used to identify trends within these types of graph. In the following scatter chart, as developed in *Creating scatter charts*, the population along the X axis is plotted against the decade along the Y axis:
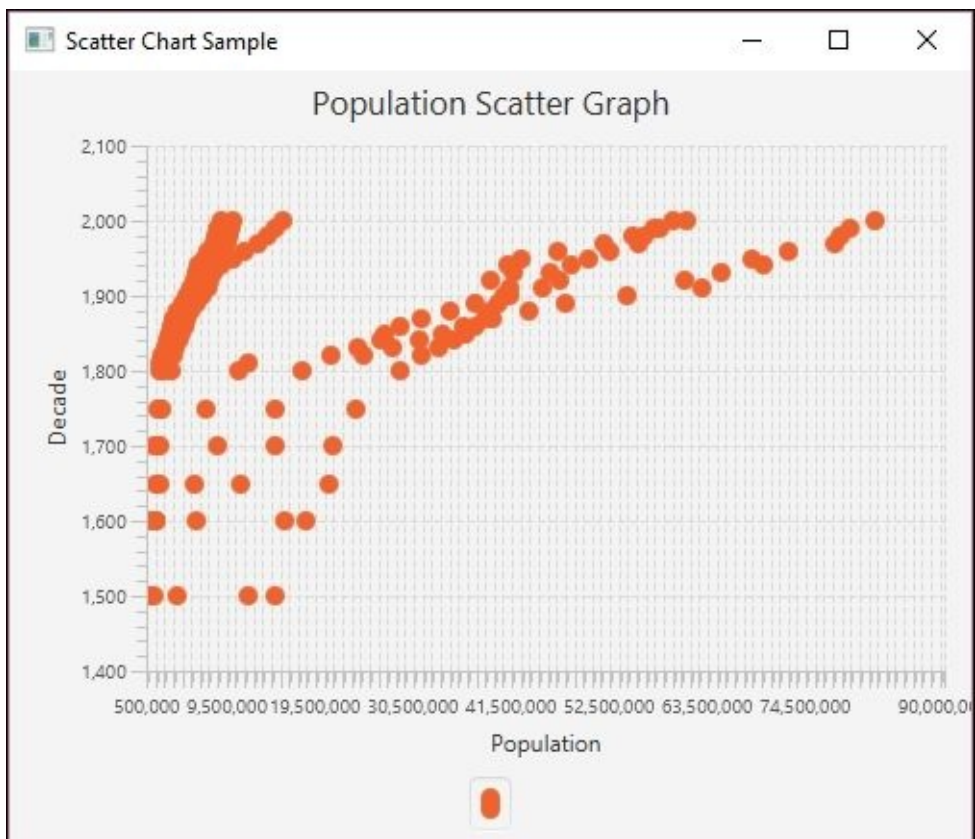
Visual analysis goals

Each type of visual expression lends itself to different types of data and data analysis purposes. One common purpose of data analysis is **data classification**. This involves determining which subset within a dataset a particular data value belongs to. This process may occur early in the data analysis process because breaking data apart into manageable and related pieces simplifies the analysis process. Often, classification is not the end goal but rather an important intermediary step before further analysis can be undertaken.

Regression analysis is a complex and important form of data analysis. It involves studying relationships between independent and dependent variables, as well as multiple independent variables. This type of statistical analysis allows the analyst to identify ranges of acceptable or expected values and determine how individual values may fit into a larger dataset. Regression analysis is a significant part of machine learning, and we will discuss it in more detail in Chapter 5, *Statisitcal Data Analysis Techniques*.

Clustering allows us to identify groups of data points within a particular set or class. While classification sorts data into similar types of datasets, clustering is concerned with the data within the set. For example, we may have a large dataset containing all feline species in the world, in the family Felidae. We could then classify these cats into two groups, Pantherinae (containing most larger cats) and Felinae (all other cats). Clustering would involve grouping subsets of similar cats within one of these classifications. For example, all tigers could be a cluster within the Pantherinae group.

Sometimes, our data analysis requires that we extract specific types of information from our dataset. The process of selecting the data to extract is known as **attribute selection** or **feature selection**. This process helps analysts simplify the data models and allows us to overcome issues with redundant or irrelevant information within our dataset.

With this introduction to basic plot and chart types, we will discuss Java support for creating these plots and charts.

Creating index charts

An index chart is a line chart that shows the percentage change of something over time. Frequently, such a chart is based on a single data attribute. In the following example, we will be using the Belgian population for six decades. The data is a subset of population data found at https://ourworldindata.org/grapher/population-by-country?tab=data:

Decade	Population
1950	8639369
1960	9118700
1970	9637800
1980	9846800
1990	9969310
2000	10263618

We start by creating the `MainApp` class, which extends `Application`. We create a series of instance variables. The `XYChart.Series` class represents a series of data points for some plot. In our case, this will be for the decades and population, which we will initialize shortly. The next declaration is for the `CategoryAxis` and `NumberAxis` instances. These represent the X and Y axes respectively. The declaration for the Y axis includes range and increment values for the population. This makes the chart a bit more readable. The last declaration is a string variable for the country:

```
public class MainApp extends Application {
    final XYChart.Series<String, Number> series =
        new XYChart.Series<>();
    final CategoryAxis xAxis = new CategoryAxis();
    final NumberAxis yAxis =
        new NumberAxis(8000000, 11000000, 1000000);
    final static String belgium = "Belgium";
    ...
}
```

In JavaFX, the `main` method usually launches the application using the base class `launch` method. Eventually, the `start` method is called, which we override. In this example, we call the

`simpleLineChart` method where the user interface is created:

```java
public static void main(String[] args) {
    launch(args);
}

public void start(Stage stage) {
    simpleIndexChart (stage);
}
```

The `simpleLineChart` follows and is passed an instance of the `Stage` class. This represents the client area of the application's window. We start by setting a title for the application and the line chart proper. The label of the *Y* axis is set. An instance of the `LineChart` class is initialized using the *X* and *Y* axis instances. This class represents the line chart:

```java
public void simpleIndexChart (Stage stage) {
    stage.setTitle("Index Chart");
    lineChart.setTitle("Belgium Population");
    yAxis.setLabel("Population");
    final LineChart<String, Number> lineChart
            = new LineChart<>(xAxis, yAxis);

    ...
}
```

The series is given a name, and then the population for each decade is added to the series using the `addDataItem` helper method:

```java
series.setName("Population");
addDataItem(series, "1950", 8639369);
addDataItem(series, "1960", 9118700);
addDataItem(series, "1970", 9637800);
addDataItem(series, "1980", 9846800);
addDataItem(series, "1990", 9969310);
addDataItem(series, "2000", 10263618);
```

The `addDataItem` method follows, which creates an `XYChart.Data` class instance using the `String` and `Number` values passed to it. It then adds the instance to the series:

```java
public void addDataItem(XYChart.Series<String, Number> series,
        String x, Number y) {
    series.getData().add(new XYChart.Data<>(x, y));
}
```

The last part of the `simpleLineChart` method creates a `Scene` class instance that represents the content of the `stage`. JavaFX uses the concept of a stage and scene to deal with the internals of the application's GUI.

The `scene` is created using a line chart, and the application's size is set to `800` by `600` pixels. The series is then added to the line chart and `scene` is added to `stage`. The `show` method displays the application:

```
Scene scene = new Scene(lineChart, 800, 600);
lineChart.getData().add(series);
stage.setScene(scene);
stage.show();
```

When the application executes the following window will be displayed:
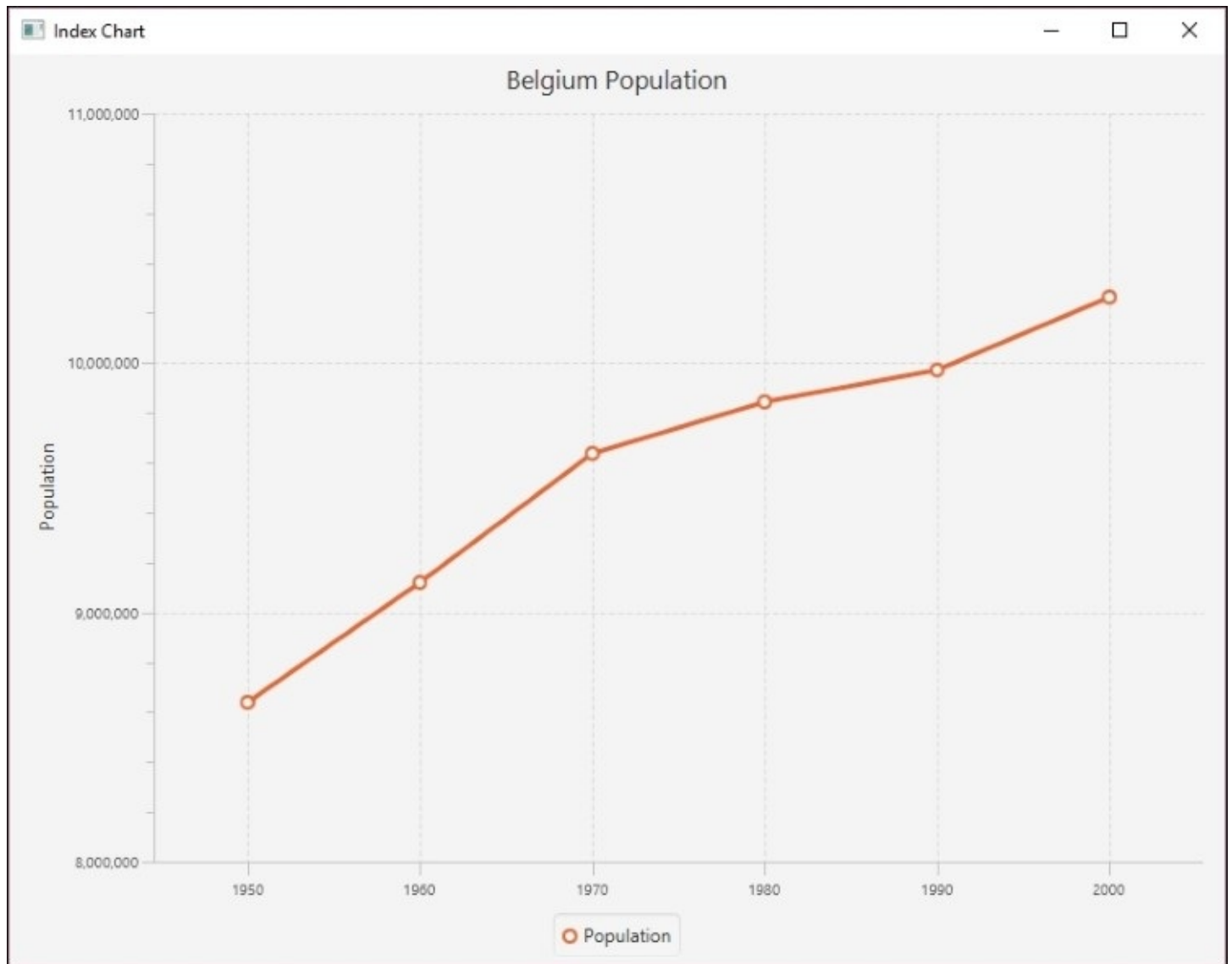
Creating bar charts

A bar chart uses two axes with rectangular bars that can be either positioned either vertically or horizontally. The length of a bar is proportional to the value it represents. A bar chart can be used to show time series data.

In the following series of examples, we will be using a set of European country populations for three decades, as listed in the following table. The data is a subset of population data found at https://ourworldindata.org/grapher/population-by-country?tab=data:

Country	1950	1960	1970
Belgium	8,639,369	9,118,700	9,637,800
France	42,518,000	46,584,000	51,918,000
Germany	68,374,572	72,480,869	77,783,164
Netherlands	10,113,527	11,486,000	13,032,335
Sweden	7,014,005	7,480,395	8,042,803
United Kingdom	50,127,000	52,372,000	55,632,000

The first of three bar charts will be constructed using JavaFX. We start with a series of declarations for the countries as part of a class that extends the `Application` class:

```
public class MainApp extends Application {
    final static String belgium = "Belgium";
    final static String france = "France";
    final static String germany = "Germany";
    final static String netherlands = "Netherlands";
    final static String sweden = "Sweden";
    final static String unitedKingdom = "United Kingdom";

...
}
```

Next, we declared a series of instance variables that represent the parts of a graph. The first are `CategoryAxis` and `NumberAxis` instances:

```
final CategoryAxis xAxis = new CategoryAxis();
```

```
final NumberAxis yAxis = new NumberAxis();
```

The population and country data is stored in a series of `XYChart.Series` instances. Here, we have declared six different series, which use a string and number pair. The first example does not use all six series, but later examples will. We will initially assign a country string and its corresponding population to three series. These series will represent the populations for the decades 1950, 1960, and 1970:

```
final XYChart.Series<String, Number> series1 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series2
    new XYChart.Series<>();
final XYChart.Series<String, Number> series3 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series4 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series5 =
    new XYChart.Series<>();
final XYChart.Series<String, Number> series6 =
    new XYChart.Series<>();
```

We will start with two simple bar charts. The first one will show the countries as categories where the year changes occur within the category on the *X* axis and the population along the *Y* axis. The second shows the decades as categories containing the counties. The last example is a stacked bar chart.

Using country as the category

The elements of the bar chart are set up in the `simpleBarChartByCountry` method. The title of the chart is set and a `BarChart` class instance is created using the two axes. The chart, its *X* axis, and its *Y* axis also have labels that are initialized here:

```
public void simpleBarChartByCountry(Stage stage) {
    stage.setTitle("Bar Chart");
    final BarChart<String, Number> barChart
                = new BarChart<>(xAxis, yAxis);
    barChart.setTitle("Country Summary");
    xAxis.setLabel("Country");
    yAxis.setLabel("Population");
    ...
}
```

Next, the first three series are initialized with a name, and then the country and population data for that series. A helper method, `addDataItem`, as introduced in the previous section, is used to add the data to each series:

```
series1.setName("1950");
addDataItem(series1,belgium, 8639369);
addDataItem(series1,france, 42518000);
addDataItem(series1,germany, 68374572);
addDataItem(series1,netherlands, 10113527);
addDataItem(series1,sweden, 7014005);
addDataItem(series1,unitedKingdom, 50127000);

series2.setName("1960");
addDataItem(series2,belgium, 9118700);
addDataItem(series2,france, 46584000);
addDataItem(series2,germany, 72480869);
addDataItem(series2,netherlands, 11486000);
addDataItem(series2,sweden, 7480395);
addDataItem(series2,unitedKingdom, 52372000);

series3.setName("1970");
addDataItem(series3,belgium, 9637800);
addDataItem(series3,france, 51918000);
addDataItem(series3,germany, 77783164);
addDataItem(series3,netherlands, 13032335);
addDataItem(series3,sweden, 8042803);
addDataItem(series3,unitedKingdom, 55632000);
```

The last part of the method creates a `scene` instance. The three series are added to the `scene` and the `scene` is attached to the `stage` using the `setScene` method. A `stage` is a class that essentially represents the client area of a window:

```
Scene scene = new Scene(barChart, 800, 600);
barChart.getData().addAll(series1, series2, series3);
stage.setScene(scene);
stage.show();
```

The last of the two methods is the `start` method, which is called automatically when the window is displayed. It is passed the `Stage` instance. Here, we call the `simpleBarChartByCountry` method:

```
public void start(Stage stage) {
    simpleBarChartByCountry(stage);
}
```

The `main` method consists of a call to the `Application` class's `launch` method:

```
public static void main(String[] args) {
    launch(args);
}
```

When the application is executed, the following graph is displayed:
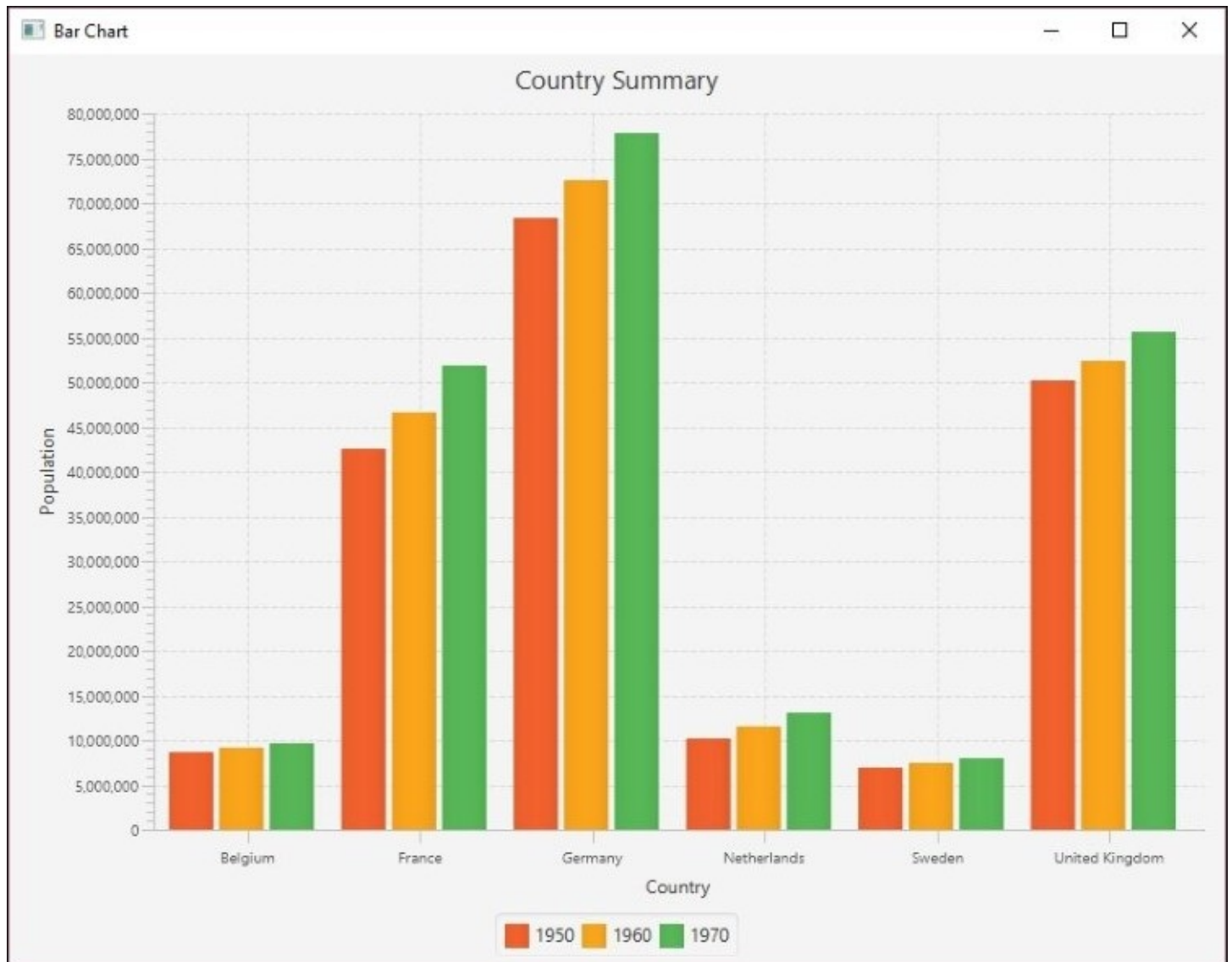
Using decade as the category

In the following example, we will demonstrate how to display the same information, but we will organize the *X* axis categories by year. We will use the `simpleBarChartByYear` method, as shown next. The axis and titles are set up in the same way as before, but with different values for the title and labels:

```
public void simpleBarChartByYear(Stage stage) {
    stage.setTitle("Bar Chart");
    final BarChart<String, Number> barChart
            = new BarChart<>(xAxis, yAxis);
    barChart.setTitle("Year Summary");
    xAxis.setLabel("Year");
    yAxis.setLabel("Population");
    ...
}
```

The following string variables are declared for the three decades:

```
String year1950 = "1950";
String year1960 = "1960";
String year1970 = "1970";
```

The data series are created in the same way as before, except the country name is used for the series name and the year is used for the category. In addition, six series are used, one for each country:

```
series1.setName(belgium);
addDataItem(series1, year1950, 8639369);
addDataItem(series1, year1960, 9118700);
addDataItem(series1, year1970, 9637800);

series2.setName(france);
addDataItem(series2, year1950, 42518000);
addDataItem(series2, year1960, 46584000);
addDataItem(series2, year1970, 51918000);

series3.setName(germany);
addDataItem(series3, year1950, 68374572);
addDataItem(series3, year1960, 72480869);
addDataItem(series3, year1970, 77783164);

series4.setName(netherlands);
addDataItem(series4, year1950, 10113527);
addDataItem(series4, year1960, 11486000);
addDataItem(series4, year1970, 13032335);

series5.setName(sweden);
addDataItem(series5, year1950, 7014005);
addDataItem(series5, year1960, 7480395);
addDataItem(series5, year1970, 8042803);

series6.setName(unitedKingdom);
```

```
addDataItem(series6, year1950, 50127000);
addDataItem(series6, year1960, 52372000);
addDataItem(series6, year1970, 55632000);
```

The scene is created and attached to the stage:

```
Scene scene = new Scene(barChart, 800, 600);
barChart.getData().addAll(series1, series2,
    series3, series4, series5, series6);
stage.setScene(scene);
stage.show();
```

The main method is unchanged, but the start method calls the simpleBarChartByYear method instead:

```
public void start(Stage stage) {
    simpleBarChartByYear(stage);
}
```

When the application is executed, the following graph is displayed:
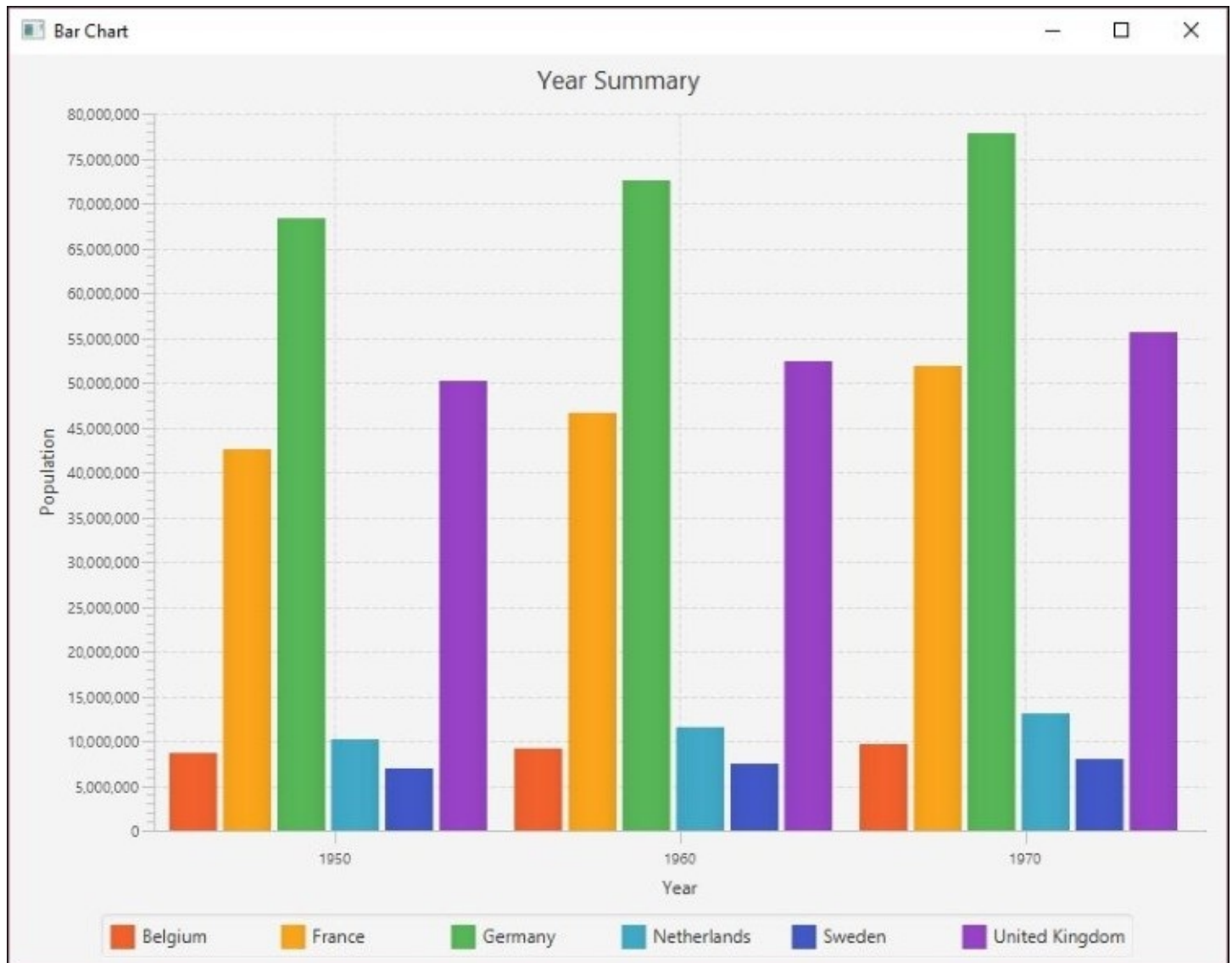
Creating stacked graphs

An area chart depicts information by allocating more space for larger values. By stacking area charts on top of each other we create a stacked graph, sometimes called a stream graph. However, stacked graphs do not work well with negative values and cannot be used for data where summation does not make sense, such as with temperatures. If too many graphs are stacked, then it can become difficult to interpret.

Next, we will show how to create a stacked bar chart. The `stackedGraphExample` method contains the code to create the bar chart. We start with familiar code to set the title and labels. However, for the *X* axis, the `setCategories` method `FXCollections.<String>observableArrayList` instance is used to set the categories. The argument of this constructor is an array of strings created by the `Arrays` class's `asList` method and the names of the countries:

```
public void stackedGraphExample(Stage stage) {
    stage.setTitle("Stacked Bar Chart");
    final StackedBarChart<String, Number> stackedBarChart
            = new StackedBarChart<>(xAxis, yAxis);
    stackedBarChart.setTitle("Country Population");
    xAxis.setLabel("Country");
    xAxis.setCategories(
        FXCollections.<String>observableArrayList(
                Arrays.asList(belgium, germany, france,
                    netherlands, sweden, unitedKingdom)));
    yAxis.setLabel("Population");
    ...
}
```

The series are initialized with the year being used for the series name and the country, and their population being added using the helper method `addDataItem`. The `scene` is then created:

```
series1.setName("1950");
addDataItem(series1, belgium, 8639369);
addDataItem(series1, france, 42518000);
addDataItem(series1, germany, 68374572);
addDataItem(series1, netherlands, 10113527);
addDataItem(series1, sweden, 7014005);
addDataItem(series1, unitedKingdom, 50127000);

series2.setName("1960");
addDataItem(series2, belgium, 9118700);
addDataItem(series2, france, 46584000);
addDataItem(series2, germany, 72480869);
addDataItem(series2, netherlands, 11486000);
addDataItem(series2, sweden, 7480395);
addDataItem(series2, unitedKingdom, 52372000);

series3.setName("1970");
addDataItem(series3, belgium, 9637800);
addDataItem(series3, france, 51918000);
```

```
addDataItem(series3, germany, 77783164);
addDataItem(series3, netherlands, 13032335);
addDataItem(series3, sweden, 8042803);
addDataItem(series3, unitedKingdom, 55632000);

Scene scene = new Scene(stackedBarChart, 800, 600);
stackedBarChart.getData().addAll(series1, series2, series3);
stage.setScene(scene);
stage.show();
```

The `main` method is unchanged, but the `start` method calls the `stackedGraphExample` method instead:

```
public void start(Stage stage) {
    stackedGraphExample(stage);
}
```

When the application is executed, the following graph is displayed:
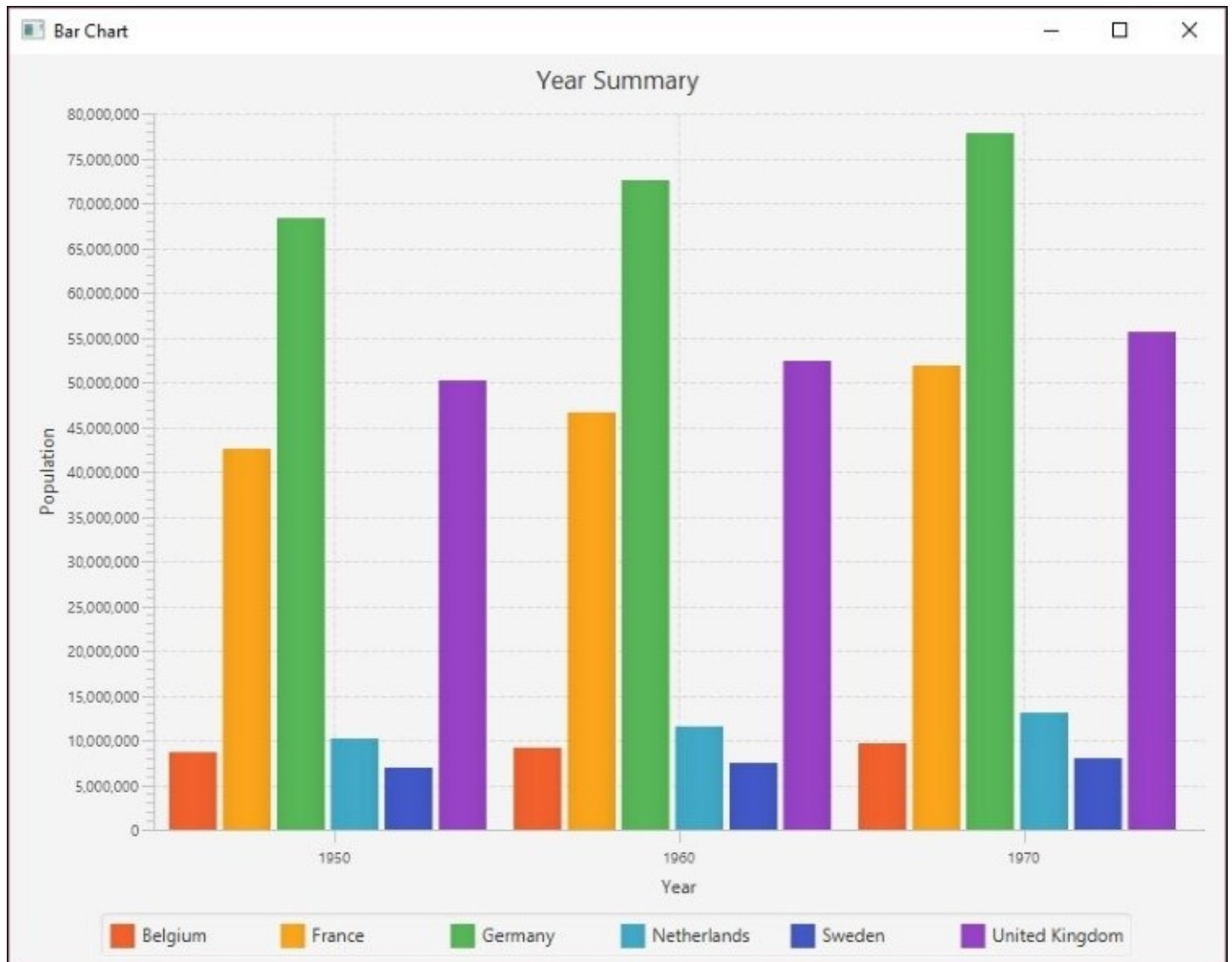
Creating pie charts

The following pie chart example is based on the 2000 population of selected European countries as summarized here:

Country	Population	Percentage
Belgium	10,263,618	3
France	61,137,000	26
Germany	82,187,909	35
Netherlands	15,907,853	7
Sweden	8,872,000	4
United Kingdom	59,522,468	25

The JavaFX implementation uses the same `Application` base class and `main` method as used in the previous examples. We will not use a separate method for creating the GUI, but instead place this code in the `start` method, as shown here:

```java
public class PieChartSample extends Application {

    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Europian Country Population");
        stage.setWidth(500);
        stage.setHeight(500);
        ...
    }

    public static void main(String[] args) {
        launch(args);
    }

}
```

A pie chart is represented by the `PieChart` class. We can create and initialize the pie chart in the constructor by using an `ObservableList` of pie chart data. This data consists of a series of `PieChart.Data` instances, each containing a text label and a percentage value.

The next sequence creates an `ObservableList` instance based on the European population data presented earlier. The `FXCollections` class's `observableArrayList` method returns an `ObservableList` instance with a list of pie chart data:

```
ObservableList<PieChart.Data> pieChartData =
        FXCollections.observableArrayList(
        new PieChart.Data("Belgium", 3),
        new PieChart.Data("France", 26),
        new PieChart.Data("Germany", 35),
        new PieChart.Data("Netherlands", 7),
        new PieChart.Data("Sweden", 4),
        new PieChart.Data("United Kingdom", 25));
```

We then create the pie chart and set its title. The pie chart is then added to the `scene`, the `scene` is associated with the `stage`, and then the window is displayed:

```
final PieChart pieChart = new PieChart(pieChartData);
pieChart.setTitle("Country Population");
((Group) scene.getRoot()).getChildren().add(pieChart);
stage.setScene(scene);
stage.show();
```

When the application is executed, the following graph is displayed:
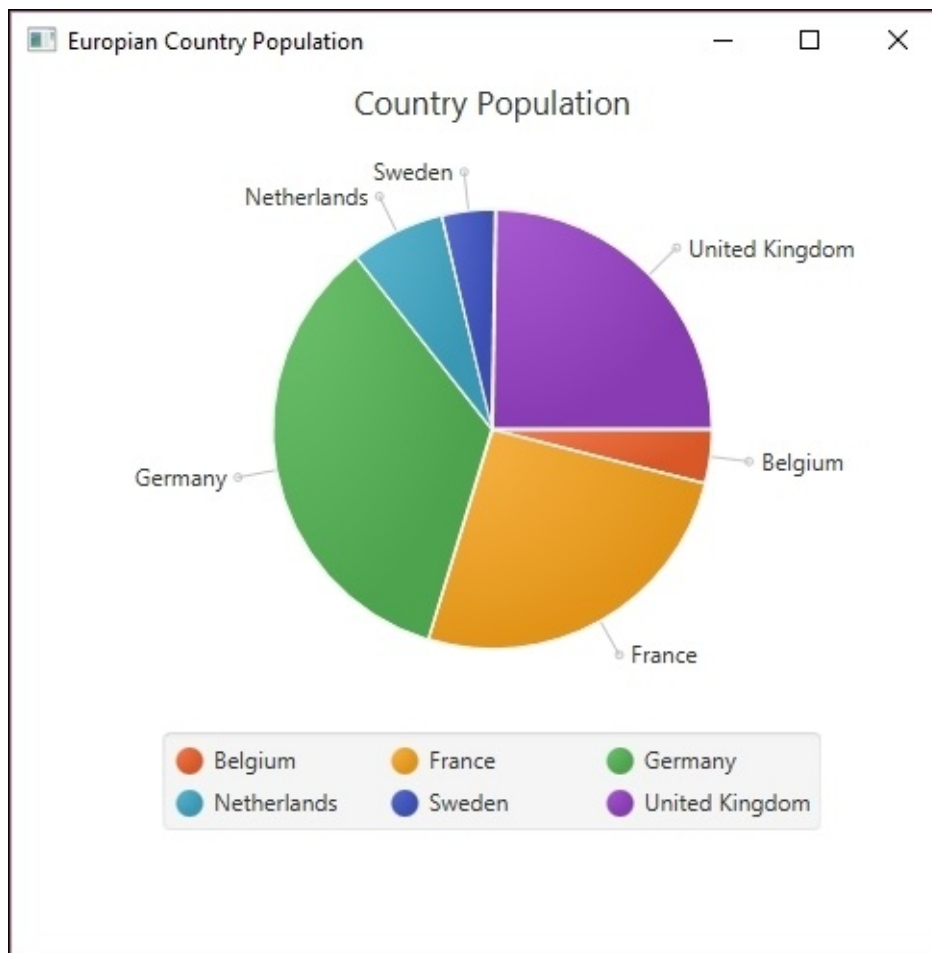
Creating scatter charts

Scatter charts also use the `XYChart.Series` class in JavaFX. For this example, we will use a set of European data that includes the previous Europeans countries and their population data for the decades 1500 through 2000. This information is stored in a file called `EuropeanScatterData.csv`. The first part of this file is shown here:

```
1500 1400000
1600 1600000
1650 1500000
1700 2000000
1750 2250000
1800 3250000
1820 3434000
1830 3750000
1840 4080000
...
```

We start with the declaration of the JavaFX `MainApp` class, as shown next. The `main` method launches the application and the `start` method creates the user interface:

```java
public class MainApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        ...
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Within the `start` method we set the title, create the axes, and create an instance of the `ScatterChart` that represents the scatter plot. The `NumberAxis` class's constructors used values that better match the data range than the default values used by its default constructor:

```java
stage.setTitle("Scatter Chart Sample");
final NumberAxis yAxis = new NumberAxis(1400, 2100, 100);
final NumberAxis xAxis = new NumberAxis(500000, 90000000,
    1000000);
final ScatterChart<Number, Number> scatterChart = new
    ScatterChart<>(xAxis, yAxis);
```

Next, the axes' labels are set along with the scatter chart's title:

```java
xAxis.setLabel("Population");
yAxis.setLabel("Decade");
scatterChart.setTitle("Population Scatter Graph");
```

An instance of the `XYChart.Series` class is created and named:

```java
XYChart.Series series = new XYChart.Series();
```

The series is populated using a `CSVReader` class instance and the file
`EuropeanScatterData.csv`. This process was discussed in Chapter 3, *Data Cleaning*:

```
try (CSVReader dataReader = new CSVReader(new
FileReader("EuropeanScatterData.csv"), ',')) {
    String[] nextLine;
    while ((nextLine = dataReader.readNext()) != null) {
            int decade = Integer.parseInt(nextLine[0]);
            int population = Integer.parseInt(nextLine[1]);
            series.getData().add(new XYChart.Data(
                population, decade));
            out.println("Decade: " + decade +
                "  Population: " + population);
    }
}
scatterChart.getData().addAll(series);
```

The JavaFX `scene` and `stage` are created, and then the plot is displayed:

```
Scene scene = new Scene(scatterChart, 500, 400);
stage.setScene(scene);
stage.show();
```

When the application is executed, the following graph is displayed:
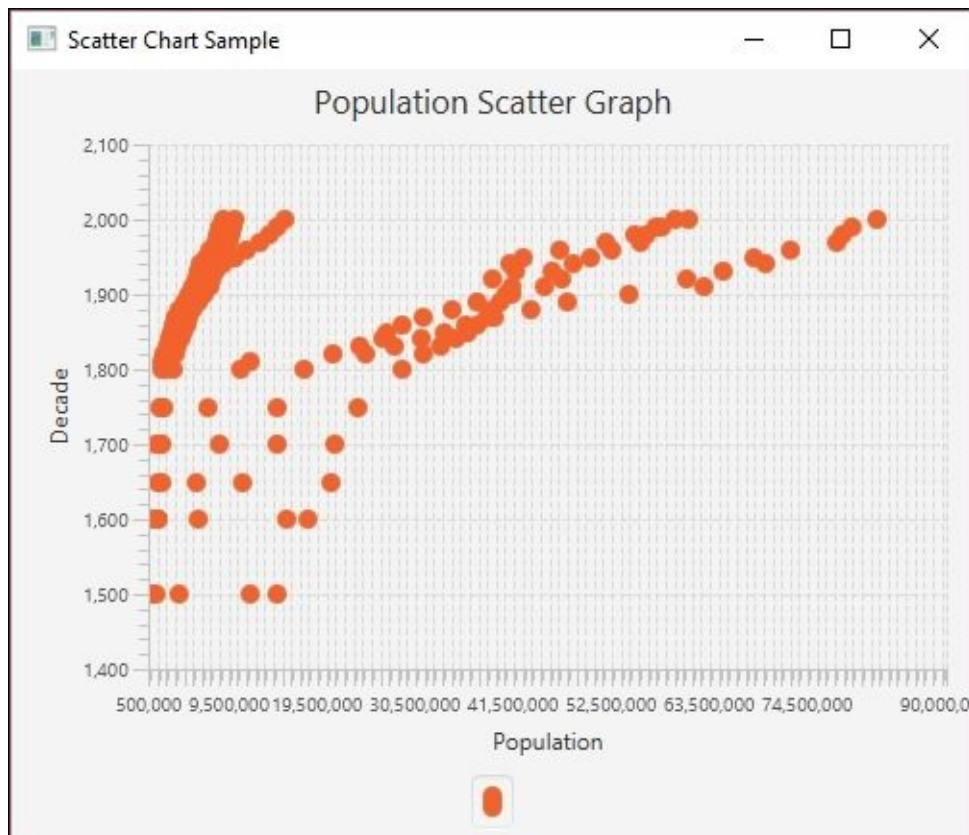
Creating histograms

Histograms, though similar in appearance to bar charts, are used to display the frequency of data items in relation to other items within the dataset. Each of the following examples using GRAL will use the `DataTable` class to initially hold the data to be displayed. In this example, we will read data from a sample file called `AgeofMarriage.csv`. This comma-separated file holds a list of ages at which people were first married.

We will create a new class, called `HistogramExample`, which extends the `JFrame` class and contains the following code within its constructor. We first create a `DataReader` object to specify that the data is in CSV format. We then use a try-catch block to handle IO exceptions and call the `DataReader` class's `read` method to place the data directly into a `DataTable` object. The first parameter of the `read` method is a `FileInputStream` object, and the second specifies the type of data expected from within the file:

```
DataReader readType=
  DataReaderFactory.getInstance().get("text/csv");
String fileName = "C://AgeofMarriage.csv";
try {
    DataTable histData = (DataTable) readType.read(
        New FileInputStream(fileName), Integer.class);
    ...
}
```

Next, we create a `Number` array to specify the ages for which we expect to have data. In this case, we expect the ages of marriage will range from `19` to `30`. We use this array to create our `Histogram` object. We include our `DataTable` from earlier and specify the orientation as well. Then we create our `DataSource`, specify our starting age, and specify the spacing along our *X* axis:

```
Number ageRange[] = {19,20,21,22,23,24,25,26,27,28,29,30};
Histogram sampleHisto = new Histogram1D(
    histData, Orientation.VERTICAL, ageRange);
DataSource sampleHistData = new EnumeratedData(sampleHisto, 19,
  1.0);
```

We use the `BarPlot` class to create our histogram from the data we read in earlier:

```
BarPlot testPlot = new BarPlot(sampleHistData);
```

The next few steps serve to format various aspects of our histogram. We use the `setInsets` method to specify how much space to place around each side of the graph within the window. We can provide a title for our graph and specify the bar width:

```
testPlot.setInsets(new Insets2D.Double(20.0, 50.0, 50.0, 20.0));
testPlot.getTitle().setText("Average Age of Marriage");
testPlot.setBarWidth(0.7);
```

We also need to format our *X* and *Y* axes. We have chosen to set our range for the *X* axis to closely match our expected age range but to provide some space on the side of the graph. Because we know the amount of sample data, we set our *Y* axis to range from `0` to `10`. In a business application, these ranges would be calculated by examining the actual dataset. We can also specify whether we want tick marks to show and where we would like the axes to intersect:

```
testPlot.getAxis(BarPlot.AXIS_X).setRange(18, 30.0);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setTickAlignment(0.0);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setTickSpacing(1);
testPlot.getAxisRenderer(BarPlot.AXIS_X).setMinorTicksVisible(false  );

testPlot.getAxis(BarPlot.AXIS_Y).setRange(0.0, 10.0);
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setTickAlignment(0.0);
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setMinorTicksVisible(false  );
testPlot.getAxisRenderer(BarPlot.AXIS_Y).setIntersection(0);
```

We also have a lot of flexibility with the color and values displayed on the graph. In this example, we have chosen to display the frequency value for each age and set our graph color to `black`:

```
PointRenderer renderHist =
    testPlot.getPointRenderers(sampleHistData).get(0);
renderHist.setColor(GraphicsUtils.deriveWithAlpha(Color.black,
  128));
renderHist.setValueVisible(true);
```

Finally, we set several properties for how we want our window to display:

```
InteractivePanel pan = new InteractivePanel(testPlot);
pan.setPannable(false);
pan.setZoomable(false);
add(pan);
setSize(1500, 700);
this.setVisible(true);
```

When the application is executed, the following graph is displayed:

Sample Histogram

Average Age of Marriage

Creating donut charts

Donut charts are similar to pie charts, but they are missing the middle section (hence the name donut). Some analysts prefer donut charts to pie charts because they do not emphasize the size of each piece within the chart and are easier to compare to other donut charts. They also provide the added advantage of taking up less space, allowing for more formatting options in the display.

In this example, we will assume our data is already populated in a two-dimensional array called `ageCount`. The first row of the array contains the possible age values, ranging again from `19` to `30` (inclusive). The second row contains the number of data values equal to each age. For example, in our dataset, there are six data values equal to `19`, so `ageCount[0][1]` contains the number six.

We create a `DataTable` and use the `add` method to add our values from the array. Notice we are testing to see if the value of a particular age is zero. In our test case, there will be zero data values equal to `23`. We are opting to add a blank space in our donut chart if there are no data values for that point. This is accomplished by using a negative number as the first parameter in the `add` method. This will set an empty space of size `3`:

```
DataTable donutData = new DataTable(Integer.class, Integer.class);
for(int Y = 0; Y < ageCount[0].length; y++){
    if(ageCount[1][y] == 0){
        donutData.add(-3, ageCount[0][y]);
    }else{
        donutData.add(ageCount[1][y], ageCount[0][y]);
    }
}
```

Next, we create our donut plot using the `PiePlot` class. We set basic properties of the plot, including specifying the values for the legend. In this case, we want our legend to reflect our age possibilities, so we use the `setLabelColumn` method to change the default labels. We also set our insets as we did in the previous example:

```
PiePlot testPlot = new PiePlot(donutData);
((ValueLegend) testPlot.getLegend()).setLabelColumn(1);
testPlot.getTitle().setText("Donut Plot Example");
testPlot.setRadius(0.9);
testPlot.setLegendVisible(true);
testPlot.setInsets(new Insets2D.Double(20.0, 20.0, 20.0, 20.0));
```

Next, we create a `PieSliceRenderer` object to set more advanced properties. Because a donut plot is basically a pie plot in essence, we will render a donut plot by calling the `setInnerRadius` method. We also specify the gap between the pie slices, the colors used, and the style of the labels:

```
PieSliceRenderer renderPie = (PieSliceRenderer)
testPlot.getPointRenderer(donutData);
```

```
renderPie.setInnerRadius(0.4);
renderPie.setGap(0.2);
LinearGradient colors = new LinearGradient(
    Color.blue, Color.green);
renderPie.setColor(colors);
renderPie.setValueVisible(true);
renderPie.setValueColor(Color.WHITE);
renderPie.setValueFont(Font.decode(null).deriveFont(Font.BOLD));
```

Finally, we create our panel and set its size:

```
add(new InteractivePanel(testPlot), BorderLayout.CENTER);
setSize(1500, 700);
setVisible(true);
```

When the application is executed, the following graph is displayed:
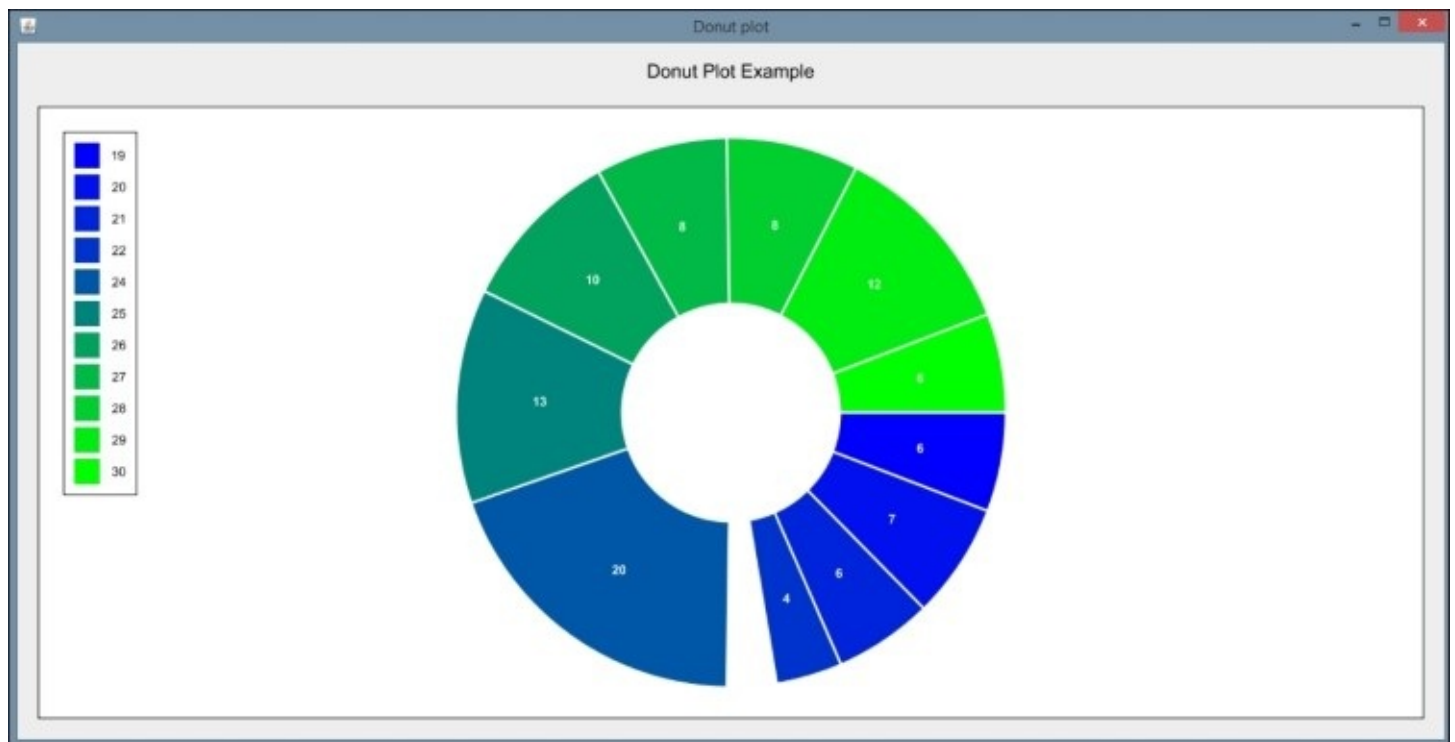
Creating bubble charts

Bubble charts are similar to scatter plots except they represent data with three dimensions. The first two dimensions are expressed on the *X* and *Y* axes and the third is represented by the size of the point plotted. This can be helpful in determining relationships between data values.

We will again use the `DataTable` class to initially hold the data to be displayed. In this example, we will read data from a sample file called `MarriageByYears.csv`. This is also a CSV file, and contains one column representing the year a marriage occurred, a second column holding the age at which a person was married, and a third column holding integers representing marital satisfaction on a scale from `1` (least satisfied) to `10` (most satisfied). We create a `DataSeries` to represent our type of desired data plot and then create a `XYPlot` object:

```
DataReader readType =
    DataReaderFactory.getInstance().get("text/csv");
String fileName = "C://MarriageByYears.csv";
try {
    DataTable bubbleData = (DataTable) readType.read(
        new FileInputStream(fileName), Integer.class,
            Integer.class, Integer.class);
DataSeries bubbleSeries = new DataSeries("Bubble", bubbleData);
XYPlot testPlot = new XYPlot(bubbleSeries);
```

Next, we set basic property information about our chart. We will set the color and turn off the vertical and horizontal grids in this example. We will also make our *X* and *Y* axes invisible in this example. Notice that we still set a range for the axes, even though they are not displayed:

```
testPlot.setInsets(new Insets2D.Double(30.0));  testPlot.setBackground(new
Color(0.75f, 0.75f, 0.75f));
XYPlotArea2D areaProp = (XYPlotArea2D) testPlot.getPlotArea();
areaProp.setBorderColor(null);
areaProp.setMajorGridX(false);
areaProp.setMajorGridY(false);
areaProp.setClippingArea(null);

testPlot.getAxisRenderer(XYPlot.AXIS_X).setShapeVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_X).setTicksVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_Y).setShapeVisible(false);
testPlot.getAxisRenderer(XYPlot.AXIS_Y).setTicksVisible(false);
testPlot.getAxis(XYPlot.AXIS_X).setRange(1940, 2020);
testPlot.getAxis(XYPlot.AXIS_Y).setRange(17, 30);
```

We can also set properties related to the bubbles drawn on the chart. Here, we set the color and shape, and specify which column of the data will be used to scale the shapes. In this case, the third column, with the marital satisfaction rating, will be used. We set it using the `setColumn` method:

```
Color color = GraphicsUtils.deriveWithAlpha(Color.black, 96);
SizeablePointRenderer renderBubble = new SizeablePointRenderer();
renderBubble.setShape(new Ellipse2D.Double(-3.5, -3.5, 4.0, 4.0));
renderBubble.setColor(color);
renderBubble.setColumn(2);
testPlot.setPointRenderers(bubbleSeries, renderBubble);
```

Finally, we create our panel and set its size:

```
add(new InteractivePanel(testPlot), BorderLayout.CENTER);
setSize(new Dimension(1500, 700));
setVisible(true);
```

When the application is executed, the following graph is displayed. Notice both the size and color of the points changes depending upon the frequency of that particular data point:
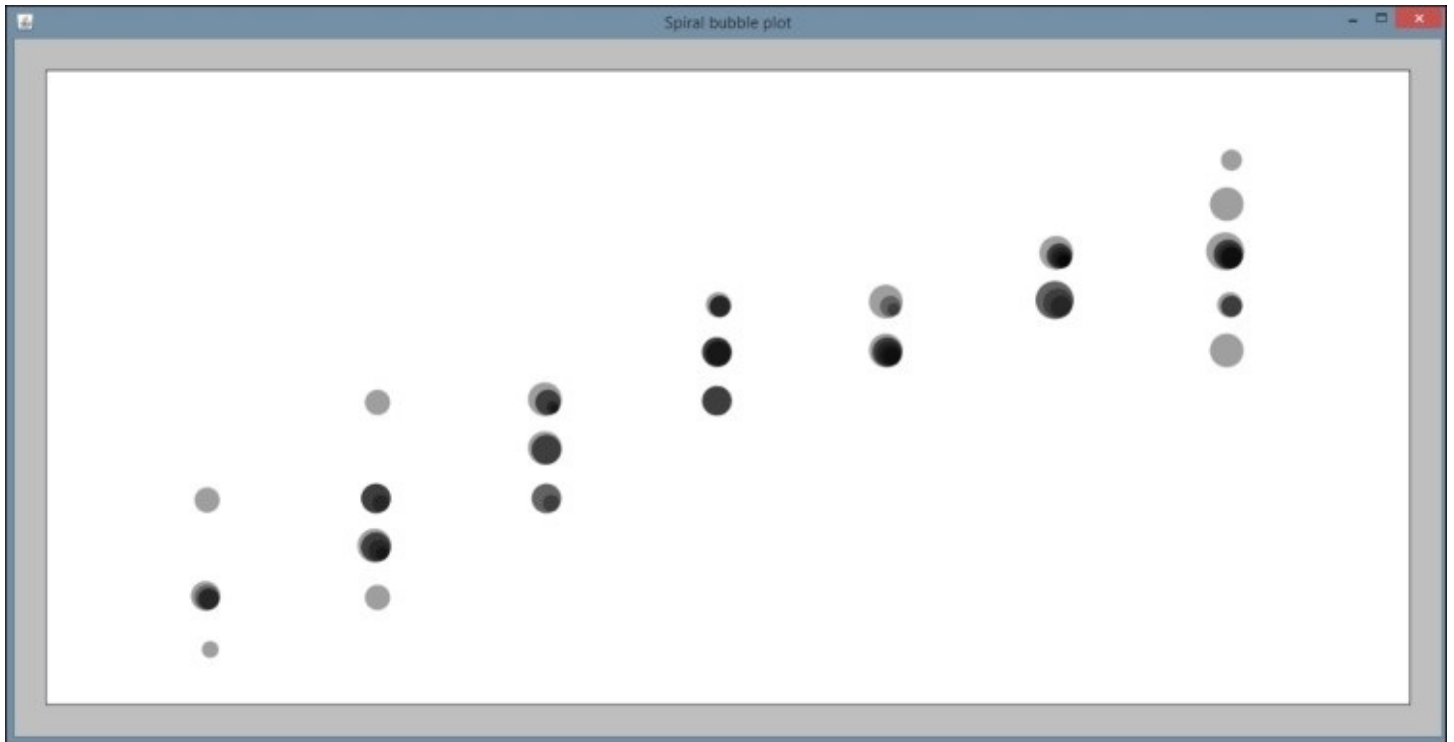
Summary

In this chapter, we introduce basic graphs, plots, and charts used to visualize data. The process of visualization enables an analyst to graphically examine the data under review. This is more intuitive, and often facilitates the rapid identification of anomalies in the data that can be hard to extract from the raw data.

Several visual representations were examined, including line charts, a variety of bar charts, pie charts, scatterplots, histograms, donut charts, and bubble charts. Each of these graphical depictions of data provides a different perspective of the data being analyzed. The most appropriate technique depends on the nature of the data being used. While we have not covered all of the possible graphical techniques, this sample provides a good overview of what is available.

We were also concerned with how Java is used to draw these graphics. Many of the examples used JavaFX. This is a readily available tool that is bundled with Java SE. However, there are several other libraries available. We used GRAL to illustrate how to generate some graphs.

With the overview of visualization techniques, we are ready to move on to other topics, where visualization will be used to better convey the essence of data science techniques. In the next chapter, we will introduce basic statistical processes, including linear regression, and we will use the techniques introduced in this chapter.

Chapter 5. Statistical Data Analysis Techniques

The intent of this chapter is not to make the reader an expert in statistical techniques. Rather, it is to familiarize the reader with the basic statistical techniques in use and demonstrate how Java can support statistical analysis. While there are quite a variety of data analysis techniques, in this chapter, we will focus on the more common tasks.

These techniques range from the relatively simple mean calculation to sophisticated regression analysis models. Statistical analysis can be a very complicated process and requires significant study to be conducted properly. We will start with an introduction to basic statistical analysis techniques, including calculating the mean, median, mode, and standard deviation of a dataset. There are numerous approaches used to calculate these values, which we will demonstrate using standard Java and third-party APIs. We will also briefly discuss sample size and hypothesis testing.

Regression analysis is an important technique for analyzing data. The technique creates a line that tries to match the dataset. The equation representing the line can be used to predict future behavior. There are several types of regression analysis. In this chapter, we will focus on simple linear regression and multiple regression. With simple linear regression, a single factor such as age is used to predict some behavior such as the likelihood of eating out. With multiple regression, multiple factors such as age, income level, and marital status may be used to predict how often a person eats out.

Predictive analytics, or analysis, is concerned with predicting future events. Many of the techniques used in this book are concerned with making predictions. Specifically, the regression analysis part of this chapter predicts future behavior.

Before we see how Java supports regression analysis, we need to discuss basic statistical techniques. We begin with mean, mode, and median.

In this chapter, we will cover the following topics:

- Working with mean, mode, and median
- Standard deviation and sample size determination
- Hypothesis testing
- Regression analysis

Working with mean, mode, and median

The mean, median, and mode are basic ways to describe characteristics or summarize information from a dataset. When a new, large dataset is first encountered, it can be helpful to know basic information about it to direct further analysis. These values are often used in later analysis to generate more complex measurements and conclusions. This can occur when we use the mean of a dataset to calculate the standard deviation, which we will demonstrate in the *Standard deviation* section of this chapter.

Calculating the mean

The term **mean,** also called the average, is computed by adding values in a list and then dividing the sum by the number of values. This technique is useful for determining the general trend for a set of numbers. It can also be used to fill in missing data elements. We are going to examine several ways to calculate the mean for a given set of data using standard Java libraries as well as third-party APIs.

Using simple Java techniques to find mean

In our first example, we will demonstrate a basic way to calculate mean using standard Java capabilities. We will use an array of `double` values called `testData`:

```
double[] testData = {12.5, 18.7, 11.2, 19.0, 22.1, 14.3, 16.9, 12.5,
    17.8, 16.9};
```

We create a `double` variable to hold the sum of all of the values and a `double` variable to hold the `mean`. A loop is used to iterate through the data and add values together. Next, the sum is divided by the `length` of our array (the total number of elements) to calculate the `mean`:

```
double total = 0;
for (double element : testData) {
    total += element;
}
double mean = total / testData.length;
out.println("The mean is " + mean);
```

Our output is as follows:

The mean is 16.19

Using Java 8 techniques to find mean

Java 8 provided additional capabilities with the introduction of optional classes. We are going to use the `OptionalDouble` class in conjunction with the `Arrays` class's `stream` method in this example. We will use the same array of doubles we used in the previous example to create an `OptionalDouble` object. If any of the numbers in the array, or the sum of the numbers in the array, is not a real number, the value of the `OptionalDouble` object will also not be a real number:

```
OptionalDouble mean = Arrays.stream(testData).average();
```

We use the `isPresent` method to determine whether we calculated a valid number for our mean. If we do not get a good result, the `isPresent` method will return `false` and we can handle any exceptions:

```
if (mean.isPresent()) {
    out.println("The mean is " + mean.getAsDouble());
} else {
```

```
    out.println("The stream was empty");
}
```

Our output is the following:

The mean is 16.19

Another, more succinct, technique using the `OptionalDouble` class involves lambda expressions and the `ifPresent` method. This method executes its argument if `mean` is a valid `OptionalDouble` object:

```
OptionalDouble mean = Arrays.stream(testData).average();
mean.ifPresent(x-> out.println("The mean is " + x));
```

Our output is as follows:

The mean is 16.19

Finally, we can use the `orElse` method to either print the mean or an alternate value if `mean` is not a valid `OptionalDouble` object:

```
OptionalDouble mean = Arrays.stream(testData).average();
out.println("The mean is " + mean.orElse(0));
```

Our output is the same:

The mean is 16.19

For our next two mean examples, we will use third-party libraries and continue using the array of doubles, `testData`.

Using Google Guava to find mean

In this example, we will use Google Guava libraries, introduced in Chapter 3, *Data Cleaning*. The `Stats` class provides functionalities for handling numeric data, including finding mean and standard deviation, which we will demonstrate later. To calculate the `mean`, we first create a `Stats` object using our `testData` array and then execute the `mean` method:

```
Stats testStat = Stats.of(testData);
double mean = testStat.mean();
out.println("The mean is " + mean);
```

Notice the difference between the default format of the output in this example.

Using Apache Commons to find mean

In our final mean examples, we use Apache Commons libraries, also introduced in Chapter 3, *Data Cleaning*. We first create a `Mean` object and then execute the `evaluate` method using our `testData`. This method returns a `double`, representing the mean of the values in the array:

```
Mean mean = new Mean();
double average = mean.evaluate(testData);
out.println("The mean is " + average);
```

Our output is the following:

The mean is 16.19

Apache Commons also provides a helpful DescriptiveStatistics class. We will use this later to demonstrate median and standard deviation, but first we will begin by calculating the mean. Using the SynchronizedDescriptiveStatistics class is advantageous as it is synchronized and therefore thread safe.

We start by creating our DescriptiveStatistics object, statTest. We then loop through our double array and add each item to statTest. We can then invoke the getMean method to calculate the mean:

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
out.println("The mean is " + statTest.getMean());
```

Our output is as follows:

The mean is 16.19

Next, we will cover the related topic: median.

Calculating the median

The mean can be misleading if the dataset contains a large number of outlying values or is otherwise skewed. When this happens, the mode and median can be useful. The term **median** is the value in the middle of a range of values. For an odd number of values, this is easy to compute. For an even number of values, the median is calculated as the average of the middle two values.

Using simple Java techniques to find median

In our first example, we will use a basic Java approach to calculate the median. For these examples, we have modified our `testData` array slightly:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5};
```

First, we use the `Arrays` class to sort our data because finding the median is simplified when the data is in numeric order:

```
Arrays.sort(testData);
```

We then handle three possibilities:

- Our list is empty
- Our list has an even number of values
- Our list has an odd number of values

The following code could be shortened, but we have been explicit to help clarify the process. If our list has an even number of values, we divide the length of the list by `2`. The first variable, `mid1`, will hold the first of two middle values. The second variable, `mid2`, will hold the second middle value. The average of these two numbers is our median value. The process for finding the median index of a list with an odd number of values is simpler and requires only that we divide the length by `2` and add `1`:

```
if(testData.length==0){      // Empty list
   out.println("No median. Length is 0");
}else if(testData.length%2==0){     // Even number of elements
   double mid1 = testData[(testData.length/2)-1];
   double mid2 = testData[testData.length/2];
   double med = (mid1 + mid2)/2;
   out.println("The median is " + med);
}else{   // Odd number of elements
   double mid = testData[(testData.length/2)+1];
   out.println("The median is " + mid);
}
```

Using the preceding array, which contains an even number of values, our output is:

The median is 16.35

To test our code for an odd number of elements, we will add the double `12.5` to the end of the array. Our new output is as follows:

The median is 16.5

Using Apache Commons to find the median

We can also calculate the median using the Apache Commons `DescriptiveStatistics` class demonstrated in the *Calculating the mean* section. We will continue using the `testData` array with the following values:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 12.5};
```

Our code is very similar to what we used to calculate the mean. We simply create our `DescriptiveStatistics` object and call the `getPercentile` method, which returns an estimate of the value stored at the percentile specified in its argument. To find the median, we use the value of `50`:

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
   statTest.addValue(num);
}
out.println("The median is " + statTest.getPercentile(50));
```

Our output is as follows:

The median is 16.2

Calculating the mode

The term **mode** is used for the most frequently occurring value in a dataset. This can be thought of as the most popular result, or the highest bar in a histogram. It can be a useful piece of information when conducting statistical analysis but it can be more complicated to calculate than it first appears. To begin, we will demonstrate a simple Java technique using the following `testData` array:

```java
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
    17.8, 16.5, 12.5};
```

We start off by initializing variables to hold the mode, the number of times the mode appears in the list, and a `tempCnt` variable. The mode and `modeCount` variables are used to hold the mode value and the number of times this value occurs in the list respectively. The variable `tempCnt` is used to count the number of times an element occurs in the list:

```java
int modeCount = 0;
double mode = 0;
int tempCnt = 0;
```

We then use nested for loops to compare each value of the array to the other values within the array. When we find matching values, we increment our `tempCnt`. After comparing each value, we test to see whether `tempCnt` is greater than `modeCount`, and if so, we change our `modeCount` and mode to reflect the new values:

```java
for (double testValue : testData){
    tempCnt = 0;
    for (double value : testData){
        if (testValue == value){
            tempCnt++;
        }
    }

    if (tempCnt > modeCount){
        modeCount = tempCnt;
        mode = testValue;
    }
}
out.println("Mode" + mode + " appears " + modeCount + " times.");
```

Using this example, our output is as follows:

The mode is 12.5 and appears 3 times.

While our preceding example seems straightforward, it poses potential problems. Modify the `testData` array as shown here, where the last entry is changed to `11.2`:

```java
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
    17.8, 16.5, 11.2};
```

When we execute our code this time, our output is as follows:

```
The mode is 12.5 and appears 2 times.
```

The problem is that our `testData` array now contains two values that appear two times each, `12.5` and `11.2`. This is known as a multimodal set of data. We can address this through basic Java code and through third-party libraries, as we will show in a moment.

However, first we will show two approaches using simple Java. The first approach will use two `ArrayList` instances and the second will use an `ArrayList` and a `HashMap` instance.

Using ArrayLists to find multiple modes

In the first approach, we modify the code used in the last example to use an `ArrayList` class. We will create two `ArrayLists`, one to hold the unique numbers within the dataset and one to hold the count of each number. We also need a `tempMode` variable, which we use next:

```
ArrayList<Integer> modeCount = new ArrayList<Integer>();
ArrayList<Double> mode = new ArrayList<Double>();
int tempMode = 0;
```

Next, we will loop through the array and test for each value in our mode list. If the value is not found in the list, we add it to `mode` and set the same position in `modeCount` to `1`. If the value is found, we increment the same position in `modeCount` by `1`:

```
for (double testValue : testData){
   int loc = mode.indexOf(testValue);
   if(loc == -1){
        mode.add(testValue);
        modeCount.add(1);
   }else{
        modeCount.set(loc, modeCount.get(loc)+1);
   }
}
```

Next, we loop through our `modeCount` list to find the largest value. This represents the mode, or the frequency of the most common value in the dataset. This allows us to select multiple modes:

```
for(int cnt = 0; cnt < modeCount.size(); cnt++){
   if (tempMode < modeCount.get(cnt)){
        tempMode = modeCount.get(cnt);
   }
}
```

Finally, we loop through our `modeCount` array again and print out any elements in mode that correspond to elements in `modeCount` containing the largest value, or mode:

```
for(int cnt = 0; cnt < modeCount.size(); cnt++){
   if (tempMode == modeCount.get(cnt)){
        out.println(mode.get(cnt) + " is a mode and appears " +
            modeCount.get(cnt) + " times.");
   }
```

```
}
```

When our code is executed, our output reflects our multimodal dataset:

```
12.5 is a mode and appears 2 times.
11.2 is a mode and appears 2 times.
```

Using a HashMap to find multiple modes

The second approach uses `HashMap`. First, we create `ArrayList` to hold possible modes, as in the previous example. We also create our `HashMap` and a variable to hold the mode:

```
ArrayList<Double> modes = new ArrayList<Double>();
HashMap<Double, Integer> modeMap = new HashMap<Double, Integer>();
int maxMode = 0;
```

Next, we loop through our `testData` array and count the number of occurrences of each value in the array. We then add the count of each value and the value itself to the `HashMap`. If the count for the value is larger than our `maxMode` variable, we set `maxMode` to our new largest number:

```
for (double value : testData) {
    int modeCnt = 0;
    if (modeMap.containsKey(value)) {
        modeCnt = modeMap.get(value) + 1;
    } else {
        modeCnt = 1;
    }
    modeMap.put(value, modeCnt);
    if (modeCnt > maxMode) {
        maxMode = modeCnt;
    }
}
```

Finally, we loop through our `HashMap` and retrieve our modes, or all values with a count equal to our `maxMode`:

```
for (Map.Entry<Double, Integer> multiModes : modeMap.entrySet()) {
    if (multiModes.getValue() == maxMode) {
        modes.add(multiModes.getKey());
    }
}
for(double mode : modes){
    out.println(mode + " is a mode and appears " + maxMode + " times.");
}
```

When we execute our code, we get the same output as in the previous example:

```
12.5 is a mode and appears 2 times.
11.2 is a mode and appears 2 times.
```

Using a Apache Commons to find multiple modes

Another option uses the Apache Commons `StatUtils` class. This class contains several methods

for statistical analysis, including multiple methods for the mean, but we will only examine the mode here. The method is named `mode` and takes an array of doubles as its parameter. It returns an array of doubles containing all modes of the dataset:

```
double[] modes = StatUtils.mode(testData);
for(double mode : modes){
    out.println(mode + " is a mode.");
}
```

One disadvantage is that we are not able to count the number of times our mode appears within this method. We simply know what the mode is, not how many times it appears. When we execute our code, we get a similar output to our previous example:

```
12.5 is a mode.
11.2 is a mode.
```

Standard deviation

Standard deviation is a measurement of how values are spread around the mean. A high deviation means that there is a wide spread, whereas a low deviation means that the values are more tightly grouped around the mean. This measurement can be misleading if there is not a single focus point or there are numerous outliers.

We begin by showing a simple example using basic Java techniques. We are using our testData array from previous examples, duplicated here:

```
double[] testData = {12.5, 18.3, 11.2, 19.0, 22.1, 14.3, 16.2, 12.5,
   17.8, 16.5, 11.2};
```

Before we can calculate the standard deviation, we need to find the average. We could use any of our techniques listed in the *Calculating the mean* section, but we will add up our values and divide by the length of testData for simplicity's sake:

```
int sum = 0;
for(double value : testData){
    sum += value;
}
double mean = sum/testData.length;
```

Next, we create a variable, sdSum, to help us calculate the standard deviation. As we loop through our array, we subtract the mean from each data value, square that value, and add it to sdSum. Finally, we divide sdSum by the length of the array and square that result:

```
int sdSum = 0;
for (double value : testData){
    sdSum += Math.pow((value - mean), 2);
}
out.println("The standard deviation is " +
Math.sqrt( sdSum / ( testData.length ) ));
```

Our output is our standard deviation:

The standard deviation is 3.3166247903554

Our next technique uses Google Guava's Stats class to calculate the standard deviation. We start by creating a Stats object with our testData. We then call the populationStandardDeviation method:

```
Stats testStats = Stats.of(testData);
double sd = testStats.populationStandardDeviation();
out.println("The standard deviation is " + sd);
```

The output is as follows:

The standard deviation is 3.3943803826056653

This example calculates the standard deviation of an entire population. Sometimes it is preferable to calculate the standard deviation of a sample subset of a population, to correct possible bias. To accomplish this, we use essentially the same code as before but replace the `populationStandardDeviation` method with `sampleStandardDeviation`:

```
Stats testStats = Stats.of(testData);
double sd = testStats.sampleStandardDeviation();
out.println("The standard deviation is " + sd);
```

In this case, our output is:

The sample standard deviation is 3.560056179332006

Our next example uses the Apache Commons `DescriptiveStatistics` class, which we used to calculate the mean and median in previous examples. Remember, this technique has the advantage of being thread safe and synchronized. After we create a `SynchronizedDescriptiveStatistics` object, we add each value from the array. We then call the `getStandardDeviation` method.

```
DescriptiveStatistics statTest =
    new SynchronizedDescriptiveStatistics();
for(double num : testData){
    statTest.addValue(num);
}
out.println("The standard deviation is " +
statTest.getStandardDeviation());
```

Notice the output matches our output from our previous example. The `getStandardDeviation` method by default returns the standard deviation adjusted for a sample:

The standard deviation is 3.5600561793320065

We can, however, continue using Apache Commons to calculate the standard deviation in either form. The `StandardDeviation` class allows you to calculate the population standard deviation or subset standard deviation. To demonstrate the differences, replace the previous code example with the following:

```
StandardDeviation sdSubset = new StandardDeviation(false);
out.println("The population standard deviation is " +
sdSubset.evaluate(testData));

StandardDeviation sdPopulation = new StandardDeviation(true);
out.println("The sample standard deviation is " +
sdPopulation.evaluate(testData));
```

On the first line, we created a new `StandardDeviation` object and set our constructor's parameter to `false`, which will produce the standard deviation of a population. The second section uses a value of `true`, which produces the standard deviation of a sample. In our example, we used the same test dataset. This means we were first treating it as though it were a subset of a

population of data. In our second example we assumed that our dataset was the entire population of data. In reality, you would might not use the same set of data with each of those methods. The output is as follows:

```
The population standard deviation is 3.3943803826056653
The sample standard deviation is 3.560056179332006
```

The preferred option will depend upon your sample and particular analyzation needs.

Sample size determination

Sample size determination involves identifying the quantity of data required to conduct accurate statistical analysis. When working with large datasets it is not always necessary to use the entire set. We use sample size determination to ensure we choose a sample small enough to manipulate and analyze easily, but large enough to represent our population of data accurately.

It is not uncommon to use a subset of data to train a model and another subset is used to test the model. This can be helpful for verifying accuracy and reliability of data. Some common consequences for a poorly determined sample size include false-positive results, false-negative results, identifying statistical significance where none exists, or suggesting a lack of significance where it is actually present. Many tools exist online for determining appropriate sample sizes, each with varying levels of complexity. One simple example is available at https://www.surveymonkey.com/mp/sample-size-calculator/.

Hypothesis testing

Hypothesis testing is used to test whether certain assumptions, or premises, about a dataset could not happen by chance. If this is the case, then the results of the test are considered to be statistically significant.

Performing hypothesis testing is not a simple task. There are many different pitfalls to avoid such as the placebo effect or the observer effect. In the former, a participant will attain a result that they think is expected. In the observer effect, also called the **Hawthorne effect**, the results are skewed because the participants know they are being watched. Due to the complex nature of human behavior analysis, some types of statistical analysis are particularly subject to skewing or corruption.

The specific methods for performing hypothesis testing are outside the scope of this book and require a solid background in statistical processes and best practices. Apache Commons provides a package, `org.apache.commons.math3.stat.inference`, with tools for performing hypothesis testing. This includes tools to perform a student's T-test, chi square, and calculating p values.

Regression analysis

Regression analysis is useful for determining trends in data. It indicates the relationship between dependent and independent variables. The independent variables determine the value of a dependent variable. Each independent variable can have either a strong or a weak effect on the value of the dependent variable. Linear regression uses a line in a scatterplot to show the trend. Non-linear regression uses some sort of curve to depict the relationships.

For example, there is a relationship between blood pressure and various factors such as age, salt intake, and **Body Mass Index** (**BMI**). The blood pressure can be treated as the dependent variable and the other factors as independent variables. Given a dataset containing these factors for a group of individuals we can perform regression analysis to see trends.

There are several types of regression analysis supported by Java. We will be examining simple linear regression and multiple linear regression. Both approaches take a dataset and derive a linear equation that best fits the data. Simple linear regression uses a single dependent and a single independent variable. Multiple linear regression uses multiple dependent variables.

There are several APIs that support simple linear regression including:

- **Apache Commons** - http://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html
- **Weka** - http://weka.sourceforge.net/doc.dev/weka/core/matrix/LinearRegression.html
- **JFree** - http://www.jfree.org/jfreechart/api/javadoc/org/jfree/data/statistics/Regression.html
- **Michael Thomas Flanagan's Java Scientific Library** - http://www.ee.ucl.ac.uk/~mflanaga/java/Regression.html

Nonlinear Java support can be found at:

- **odinsbane**/**least-squares-in-java** - https://github.com/odinsbane/least-squares-in-java
- **NonLinearLeastSquares** (**Parallel Java Library Documentation**) - https://www.cs.rit.edu/~ark/pj/doc/edu/rit/numeric/NonLinearLeastSquares.html

There are several statistics that evaluate the effectiveness of an analysis. We will focus on basic statistics.

Residuals are the difference between the actual data values and the predicted values. The **Residual Sum of Squares** (**RSS**) is the sum of the squares of residuals. Essentially it measures the discrepancy between the data and a regression model. A small RSS indicates the model closely matches the data. RSS is also known as the **Sum of Squared Residuals** (**SSR**) or the **Sum of Squared Errors** (**SSE**) of prediction.

The **Mean Square Error** (**MSE**) is the sum of squared residuals divided by the degrees of freedom. The number of degrees of freedom is the number of independent observations (N) minus

the number of estimates of population parameters. For simple linear regression this $N - 2$ because there are two parameters. For multiple linear regression it depends on the number of independent variables used.

A small MSE also indicates that the model fits the dataset well. You will see both of these statistics used when discussing linear regression models.

The correlation coefficient measures the association between two variables of a regression model. The correlation coefficient ranges from *-1* to *+1*. A value of *+1* means that two variables are perfectly related. When one increases, so does the other. A correlation coefficient of *-1* means that two variables are negatively related. When one increases, the other decreases. A value of 0 means there is no correlation between the variables. The coefficient is frequently designated as R. It will often be squared, thus ignoring the sign of the relation. The Pearson's product moment correlation coefficient is normally used.

Using simple linear regression

Simple linear regression uses a least squares approach where a line is computed that minimizes the sum of squared of the distances between the points and the line. Sometimes the line is calculated without using the *Y* intercept term. The regression line is an estimate. We can use the line's equation to predict other data points. This is useful when we want to predict future events based on past performance.

In the following example we use the Apache Commons SimpleRegression class with the Belgium population dataset used in Chapter 4, *Data Visualization*. The data is duplicated here for your convenience:

Decade	Population
1950	8639369
1960	9118700
1970	9637800
1980	9846800
1990	9969310
2000	10263618

While the application that we will demonstrate is a JavaFX application, we will focus on the linear regression aspects of the application. We used a JavaFX program to generate a chart to show the regression results.

The body of the `start` method follows. The input data is stored in a two-dimension array as shown here:

```
double[][] input = {{1950, 8639369}, {1960, 9118700},
    {1970, 9637800}, {1980, 9846800}, {1990, 9969310},
    {2000, 10263618}};
```

An instance of the `SimpleRegression` class is created and the data is added using the `addData` method:

```
SimpleRegression regression = new SimpleRegression();
regression.addData(input);
```

We will use the model to predict behavior for several years as declared in the array that follows:

```
double[] predictionYears = {1950, 1960, 1970, 1980, 1990, 2000,
    2010, 2020, 2030, 2040};
```

We will also format our output using the following `NumberFormat` instances. One is used for the year where the `setGroupingUsed` method with a parameter of false suppresses commas.

```
NumberFormat yearFormat = NumberFormat.getNumberInstance();
yearFormat.setMaximumFractionDigits(0);
yearFormat.setGroupingUsed(false);
NumberFormat populationFormat = NumberFormat.getNumberInstance();
populationFormat.setMaximumFractionDigits(0);
```

The `SimpleRegression` class possesses a `predict` method that is passed a value, a year in this case, and returns the estimated population. We use this method in a loop and call the method for each year:

```
for (int i = 0; i < predictionYears.length; i++) {
    out.println(nf.format(predictionYears[i]) + "-"
            + nf.format(regression.predict(predictionYears[i])));
}
```

When the program is executed, we get the following output:

```
1950-8,801,975
1960-9,112,892
1970-9,423,808
1980-9,734,724
1990-10,045,641
2000-10,356,557
2010-10,667,474
2020-10,978,390
2030-11,289,307
2040-11,600,223
```

To see the results graphically, we generated the following index chart. The line matches the actual population values fairly well and shows the projected populations in the future.

Simple Linear Regression

The `SimpleRegession` class supports a number of methods that provide additional information about the regression. These methods are summarized next:

Method	Meaning
getR	Returns Pearson's product moment correlation coefficient
getRSquare	Returns the coefficient of determination (R-square)
getMeanSquareError	Returns the MSE

getSlope	The slope of the line	
getIntercept	The intercept	

We used the helper method, `displayAttribute`, to display various attribute values as shown here:

```
displayAttribute(String attribute, double value) {
    NumberFormat numberFormat = NumberFormat.getNumberInstance();
    numberFormat.setMaximumFractionDigits(2);
    out.println(attribute + ": " + numberFormat.format(value));
}
```

We called the previous methods for our model as shown next:

```
displayAttribute("Slope",regression.getSlope());
displayAttribute("Intercept", regression.getIntercept());
displayAttribute("MeanSquareError",
    regression.getMeanSquareError());
displayAttribute("R", + regression.getR());
displayAttribute("RSquare", regression.getRSquare());
```

The output follows:

```
Slope: 31,091.64
Intercept: -51,826,728.48
MeanSquareError: 24,823,028,973.4
R: 0.97
RSquare: 0.94
```

As you can see, the model fits the data nicely.

Using multiple regression

Our intent is not to provide a detailed explanation of multiple linear regression as that would be beyond the scope of this section. A more through treatment can be found at http://www.biddle.com/documents/bcg_comp_chapter4.pdf. Instead, we will explain the basics of the approach and show how we can use Java to perform multiple regression.

Multiple regression works with data where multiple independent variables exist. This happens quite often. Consider that the fuel efficiency of a car can be dependent on the octane level of the gas being used, the size of the engine, the average cruising speed, and the ambient temperature. All of these factors can influence the fuel efficiency, some to a larger degree than others.

The independent variable is normally represented as Y where there are multiple dependent variables represented using different Xs. A simplified equation for a regression using three dependent variables follows where each variable has a coefficient. The first term is the intercept. These coefficients are not intended to represent real values but are only used for illustrative purposes.

Y = 11 + 0.75 X1 + 0.25 X2 − 2 X3

The intercept and coefficients are generated using a multiple regression model based on sample data. Once we have these values, we can create an equation to predict other values.

We will use the Apache Commons `OLSMultipleLinearRegression` class to perform multiple regression using cigarette data. The data has been adapted from http://www.amstat.org/publications/jse/v2n1/datasets.mcintyre.html. The data consists of 25 entries for different brands of cigarettes with the following information:

- Brand name
- Tar content (mg)
- Nicotine content (mg)
- Weight (g)
- Carbon monoxide content (mg)

The data has been stored in a file called `data.csv` as shown in the following partial listing of its contents where the columns values match the order of the previous list:

```
Alpine,14.1,.86,.9853,13.6
Benson&Hedges,16.0,1.06,1.0938,16.6
BullDurham,29.8,2.03,1.1650,23.5
CamelLights,8.0,.67,.9280,10.2
...
```

The following is a scatter plot chart showing the relationship of the data:
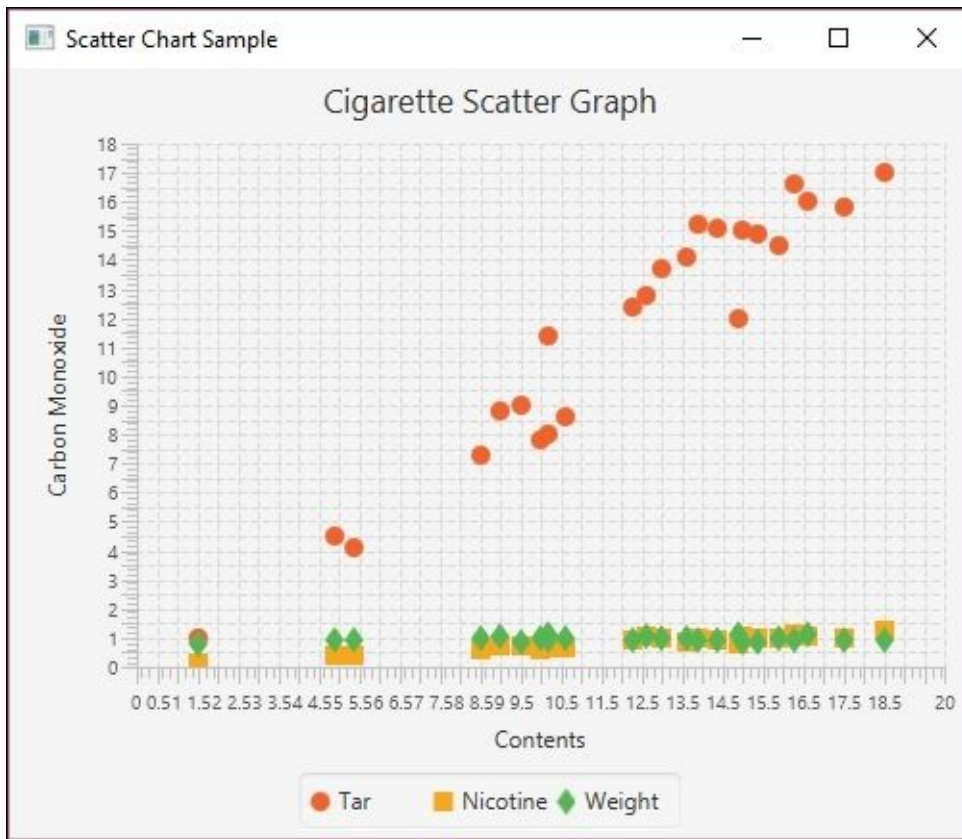
Multiple Regression Scatter plot

We will use a JavaFX program to create the scatter plot and to perform the analysis. We start with the `MainApp` class as shown next. In this example we will focus on the multiple regression code and we do not include the JavaFX code used to create the scatter plot. The complete program can be downloaded from http://www.packtpub.com/support.

The data is held in a one-dimensional array and a `NumberFormat` instance will be used to format the values. The array size reflects the *25* entries and the *4* values per entry. We will not be using the brand name in this example.

```
public class MainApp extends Application {
    private final double[] data = new double[100];
    private final NumberFormat numberFormat =
        NumberFormat.getNumberInstance();
    ...
    public static void main(String[] args) {
        launch(args);
    }
}
```

The data is read into the array using a `CSVReader` instance as shown next:

```
int i = 0;
try (CSVReader dataReader = new CSVReader(
```

```
            new FileReader("data.csv"), ',')) {
    String[] nextLine;
    while ((nextLine = dataReader.readNext()) != null) {
        String brandName = nextLine[0];
        double tarContent = Double.parseDouble(nextLine[1]);
        double nicotineContent = Double.parseDouble(nextLine[2]);
        double weight = Double.parseDouble(nextLine[3]);
        double carbonMonoxideContent =
            Double.parseDouble(nextLine[4]);
        data[i++] = carbonMonoxideContent;
        data[i++] = tarContent;
        data[i++] = nicotineContent;
        data[i++] = weight;
        ...
    }
}
```

Apache Commons possesses two classes that perform multiple regression:

- `OLSMultipleLinearRegression`- **Ordinary Least Square (OLS)** regression
- `GLSMultipleLinearRegression`- **Generalized Least Squared (GLS)** regression

When the latter technique is used, the correlation within elements of the model impacts the results addversely. We will use the `OLSMultipleLinearRegression` class and start with its instantiation:

```
OLSMultipleLinearRegression ols =
new OLSMultipleLinearRegression();
```

We will use the `newSampleData` method to initialize the model. This method needs the number of observations in the dataset and the number of independent variables. It may throw an `IllegalArgumentException` exception which needs to be handled.

```
int numberOfObservations = 25;
int numberOfIndependentVariables = 3;
try {
    ols.newSampleData(data, numberOfObservations,
        numberOfIndependentVariables);
} catch (IllegalArgumentException e) {
     // Handle exceptions
}
```

Next, we set the number of digits that will follow the decimal point to two and invoke the `estimateRegressionParameters` method. This returns an array of values for our equation, which are then displayed:

```
numberFormat.setMaximumFractionDigits(2);
double[] parameters = ols.estimateRegressionParameters();
for (int i = 0; i < parameters.length; i++) {
    out.println("Parameter " + i +": " +
        numberFormat.format(parameters[i]));
}
```

When executed we will get the following output which gives us the parameters needed for our regression equation:

```
Parameter 0: 3.2
Parameter 1: 0.96
Parameter 2: -2.63
Parameter 3: -0.13
```

To predict a new dependent value based on a set of independent variables, the getY method is declared, as shown next. The parameters parameter contains the generated equation coefficients. The arguments parameter contains the value for the dependent variables. These are used to calculate the new dependent value which is returned:

```
public double getY(double[] parameters, double[] arguments) {
    double result = 0;
    for(int i=0; i<parameters.length; i++) {
        result += parameters[i] * arguments[i];
    }
    return result;
}
```

We can test this method by creating a series of independent values. Here we used the same values as used for the SalemUltra entry in the data file:

```
double arguments1[] = {1, 4.5, 0.42, 0.9106};
out.println("X: " + 4.9 + "  y: " +
        numberFormat.format(getY(parameters,arguments1)));
```

This will give us the following values:

```
X: 4.9 y: 6.31
```

The return value of 6.31 is different from the actual value of 4.9. However, using the values for VirginiaSlims:

```
double arguments2[] = {1, 15.2, 1.02, 0.9496};
out.println("X: " + 13.9 + "  y: " +
        numberFormat.format(getY(parameters,arguments2)));
```

We get the following result:

```
X: 13.9 y: 15.03
```

This is close to the actual value of 13.9. Next, we use a different set of values than found in the dataset:

```
double arguments3[] = {1, 12.2, 1.65, 0.86};
out.println("X: " + 9.9 + "  y: " +
        numberFormat.format(getY(parameters,arguments3)));
```

The result follows:

```
X: 9.9 y: 10.49
```

The values differ but are still close. The following figure shows the predicted data in relation to the original data:

Multiple regression projected

The `OLSMultipleLinearRegression` class also possesses several methods to evaluate how well the models fits the data. However, due to the complex nature of multiple regression, we have not discussed them here.

Summary

In this chapter, we provided a brief introduction to the basic statistical analysis techniques you may encounter in data science applications. We started with simple techniques to calculate the mean, median, and mode of a set of numerical data. Both standard Java and third-party Java APIs were used to show how these attributes are calculated. While these techniques are relatively simple, there are some issues that need to be considered when calculating them.

Next, we examined linear regression. This technique is more predictive in nature and attempts to calculate other values in the future or past based on a sample dataset. We examine both simple linear regression and multiple regression and used Apache Commons classes to perform the regression and JavaFX to draw graphs.

Simple linear regression uses a single independent variable to predict a dependent variable. Multiple regression uses more than one independent variable. Both of these techniques have statistical attributes used to assess how well they match the data.

We demonstrated the use of the Apache Commons `OLSMultipleLinearRegression` class to perform multiple regression using cigarette data. We were able to use multiple attributes to create an equation that predicts the carbon monoxide output.

With these statistical techniques behind us, we can now examine basic machine learning techniques in the next chapter. This will include a detailed discussion of multilayer perceptrons and various other neural networks.

Chapter 6. Machine Learning

Machine learning is a broad topic with many different supporting algorithms. It is generally concerned with developing techniques that allow applications to learn without having to be explicitly programmed to solve a problem. Typically, a model is built to solve a class of problems and then is trained using sample data from the problem domain. In this chapter, we will address a few of the more common problems and models used in data science.

Many of these techniques use training data to teach a model. The data consists of various representative elements of the problem space. Once the model has been trained, it is tested and evaluated using testing data. The model is then used with input data to make predictions.

For example, the purchases made by customers of a store can be used to train a model. Subsequently, predictions can be made about customers with similar characteristics. Due to the ability to predict customer behavior, it is possible to offer special deals or services to entice customers to return or facilitate their visit.

There are several ways of classifying machine learning techniques. One approach is to classify them according to the learning style:

- **Supervised learning**: With supervised learning, the model is trained with data that matches input characteristic values to the correct output values
- **Unsupervised learning**: In unsupervised learning, the data does not contain results, but the model is expected to determine relationships on its own.
- **Semi-supervised**: This technique uses a small amount of labeled data containing the correct response with a larger amount of unlabeled data. The combination can lead to improved results.
- **Reinforcement learning**: This is similar to supervised learning but a reward is provided for good results.
- **Deep learning**: This approach models high-level abstractions using a graph that contains multiple processing levels.

In this chapter, we will only be able to touch on a few of these techniques. Specifically, we will illustrate three techniques that use supervised learning:

- **Decision trees**: A tree is constructed using the features of the problem as internal nodes and the results as leaves
- **Support vector machines**: Generally used for classification by creating a hyperplane that separates the dataset and then making predictions
- **Bayesian networks**: Models used to depict probabilistic relationships between events within an environment

For unsupervised learning, we will show how **association rule learning** can be used to find relationships between elements of a dataset. However, we will not address unsupervised learning in this chapter.

We will discuss the elements of reinforcement learning and discuss a few specific variations of this technique. We will also provide links to resources for further exploration.

The discussion of deep learning is postponed to Chapter 8, *Deep Learning*. This technique builds upon neural networks, which will be discussed in Chapter 7, *Neural Networks*.

In this chapter, we will cover the following specific topics:

- Decision trees
- Support vector machines
- Bayesian networks
- Association rule learning
- Reinforcement learning

Supervised learning techniques

There are a large number of supervised machine learning algorithms available. We will examine three of them: decision trees, support vector machines, and Bayesian networks. They all use annotated datasets that contain attributes and a correct response. Typically, a training and a testing dataset is used.

We start with a discussion of decision trees.

Decision trees

A machine learning **decision tree** is a model used to make predictions. It effectively maps certain observations to conclusions about a target. The term **tree** comes from the branches that reflect different states or values. The leaves of a tree represent results and the branches represent features that lead to the results. In data mining, a decision tree is a description of data used for classification. For example, we can use a decision tree to determine whether an individual is likely to buy an item based on certain attributes such as income level and postal code.

We want to create a decision tree that predicts results based on other variables. When the target variable takes on continuous values such as real numbers, the tree is called a **regression tree**.

A tree consists of internal nodes and leaves. Each internal node represents a feature of the model such as the number of years of education or whether a book is paperback or hardcover. The edges leading out of an internal node represent the values of these features. Each leaf is called a **class** and has an **associated probability distribution**.

For example, we will be using a dataset that deals with the success of a book based on its binding type, use of color, and genre. One possible decision tree based on this dataset follows:
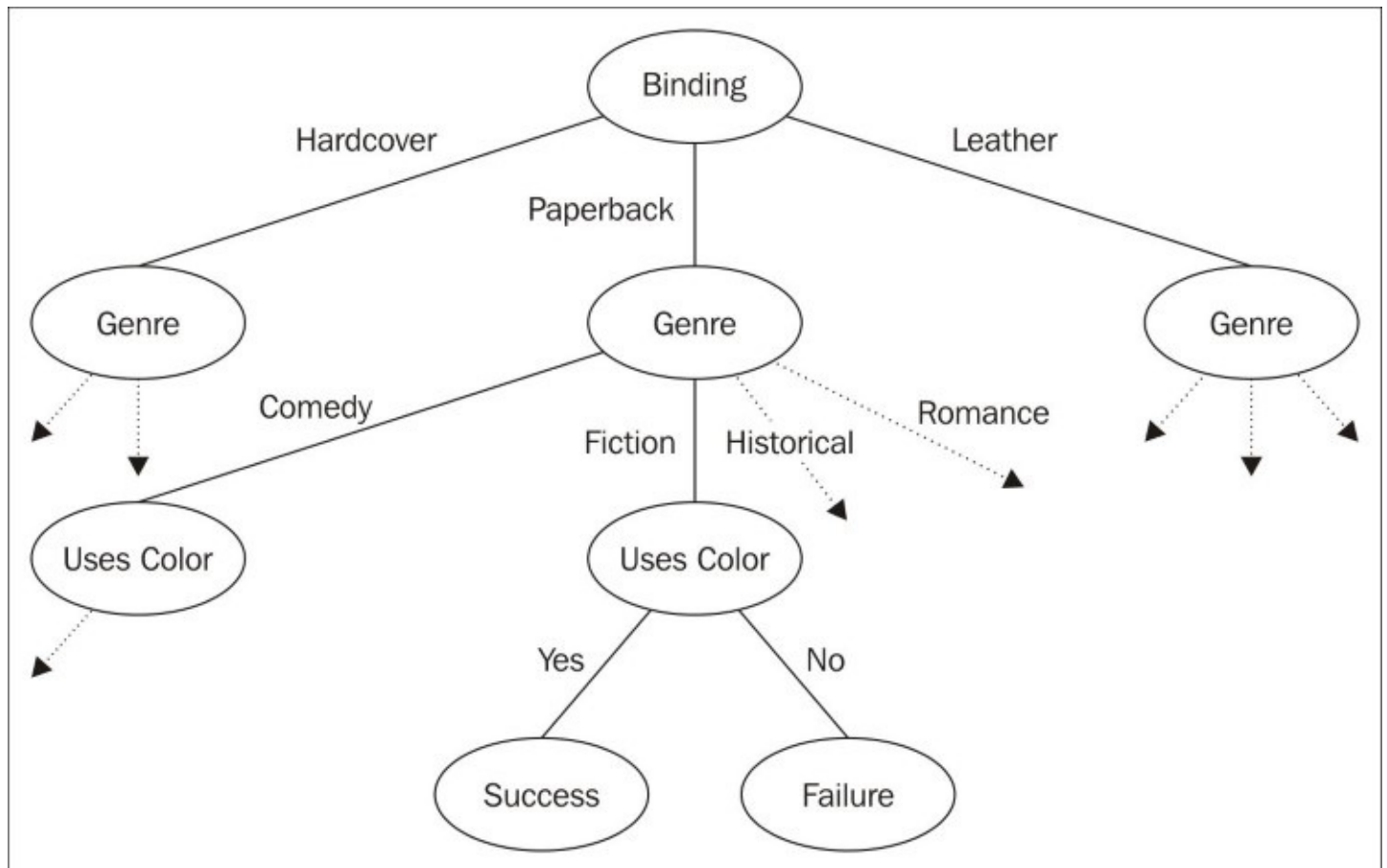
Decision tree

Decision trees are useful and easy to understand. Preparing data for a model is straightforward even for large datasets.

Decision tree types

A tree can be *taught* by dividing an input dataset by the features. This is often done in a recursive fashion and is called **recursive partitioning** or **Top-Down Induction of Decision Trees** (**TDIDT**). The recursion is bounded when node's values are all of the same type as the target or the recursion no longer adds value.

Classification and Regression Tree (**CART**) analysis refers to two different types of decision tree types:

- **Classification tree analysis**: The leaf corresponds to a target feature
- **Regression tree analysis**: The leaf possesses a real number representing a feature

During the process of analysis, multiple trees may be created. There are several techniques used to create trees. The techniques are called **ensemble methods**:

- **Bagging decision trees**: The data is resampled and frequently used to obtain a prediction based on consensus
- **Random forest classifier**: Used to improve the classification rate
- **Boosted trees**: This can be used for regression or classification problems
- **Rotation forest**: Uses a technique called **Principal Component Analysis** (**PCA**)

With a given set of data, it is possible that more than one tree models the data. For example, the root of a tree may specify whether a bank has an ATM machine and a subsequent internal node may specify the number of tellers. However, the tree could be created where the number of tellers is at the root and the existence of an ATM is an internal node. The difference in the structure of the tree can determine how efficient the tree is.

There are a number of ways of determining the order of the nodes of a tree. One technique is to select an attribute that provides the most information gain; that is, choose an attribute that better helps narrow down the possible decisions fastest.

Decision tree libraries

There are several Java libraries that support decision trees:

- **Weka**: http://www.cs.waikato.ac.nz/ml/weka/
- **Apache Spark**: https://spark.apache.org/docs/1.2.0/mllib-decision-tree.html
- **JBoost**: http://jboost.sourceforge.net
- **MAchine Learning for LanguagE Toolkit** (**MALLET**): http://mallet.cs.umass.edu

We will use **Waikato Environment for Knowledge Analysis (Weka)** to demonstrate how to create a decision tree in Java. Weka is a tool that has a GUI interface that permits analysis of data. It can also be invoked from the command line or through a Java API, which we will use.

As a tree is being built, a variable is selected to split the tree. There are several techniques used to select a variable. The one we use is determined by how much information is gained by choosing a variable. Specifically, we will use the **C4.5 algorithm** as supported by Weka's `J48` class.

Weka uses an `.arff` file to hold a dataset. This file is human readable and consists of two sections. The first is a header section; it describes the data in the file. This section uses the ampersand to specify the relation and attributes of the data. The second section is the data section; it consists of a comma-delimited set of data.

Using a decision tree with a book dataset

For this example, we will use a file called `books.arff`. It is shown next and uses four features called attributes. The features specify how a book is bound, whether it uses multiple colors, its genre, and a result indicating whether the book was purchased or not. The header section is shown here:

```
@RELATION book_purchases
@ATTRIBUTE Binding {Hardcover, Paperback, Leather}
@ATTRIBUTE Multicolor {yes, no}
@ATTRIBUTE Genre {fiction, comedy, romance, historical}
@ATTRIBUTE Result {Success, Failure}
```

The data section follows and consists of 13 book entries:

```
@DATA
Hardcover,yes,fiction,Success
Hardcover,no,comedy,Failure
Hardcover,yes,comedy,Success
Leather,no,comedy,Success
Leather,yes,historical,Success
Paperback,yes,fiction,Failure
Paperback,yes,romance,Failure
Leather,yes,comedy,Failure
Paperback,no,fiction,Failure
Paperback,yes,historical,Failure
Hardcover,yes,historical,Success
Paperback,yes,comedy,Success
Hardcover,yes,comedy,Success
```

We will use the `BookDecisionTree` class as defined next to process this file. It uses one constructor and three methods:

- `BookDecisionTree`: Reads in the trainer data and creates an `Instance` object used to process the data
- `main`: Drives the application
- `performTraining`: Trains the model using the dataset
- `getTestInstance`: Creates a test case

The `Instances` class holds elements representing the individual dataset elements:

```java
public class BookDecisionTree {
    private Instances trainingData;

    public static void main(String[] args) {
        ...
    }

    public BookDecisionTree(String fileName) {
        ...
    }

    private J48 performTraining() {
        ...
    }

    private Instance getTestInstance(
        ...
    }
}
```

The constructor opens a file and uses the `BufferReader` instance to create an instance of the `Instances` class. Each element of the dataset will be either a feature or a result. The `setClassIndex` method specifies the index of the result class. In this case, it is the last index of the dataset and corresponds to success or failure:

```java
public BookDecisionTree(String fileName) {
    try {
        BufferedReader reader = new BufferedReader(
            new FileReader(fileName));
        trainingData = new Instances(reader);
        trainingData.setClassIndex(
            trainingData.numAttributes() - 1);
    } catch (IOException ex) {
        // Handle exceptions
    }
}
```

We will use the `J48` class to generate a decision tree. This class uses the C4.5 decision tree algorithm for generating a pruned or unpruned tree. The `setOptions` method specifies that an unpruned tree be used. The `buildClassifier` method actually creates the classifier based on the dataset used:

```java
private J48 performTraining() {
    J48 j48 = new J48();
    String[] options = {"-U"};
    try {
        j48.setOptions(options);
        j48.buildClassifier(trainingData);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return j48;
}
```

We will want to test the model, so we will create an object that implements the `Instance` interface for each test case. A `getTestInstance` helper method is passed three arguments representing the three features of a data element. The `DenseInstance` class is a class that implements the `Instance` interface. The values passed are assigned to the instance and the instance is returned:

```
private Instance getTestInstance(
        String binding, String multicolor, String genre) {
    Instance instance = new DenseInstance(3);
    instance.setDataset(trainingData);
    instance.setValue(trainingData.attribute(0), binding);
    instance.setValue(trainingData.attribute(1), multicolor);
    instance.setValue(trainingData.attribute(2), genre);
    return instance;
}
```

The `main` method uses all the previous methods to process and test our book dataset. First, a `BookDecisionTree` instance is created using the name of the book dataset file:

```
public static void main(String[] args) {
    try {
        BookDecisionTree decisionTree =
            new BookDecisionTree("books.arff");
        ...
    } catch (Exception ex) {
        // Handle exceptions
    }
}
```

Next, the `performTraining` method is invoked to train the model. We also display the tree:

```
J48 tree = decisionTree.performTraining();
System.out.println(tree.toString());
```

When executed, the following will be displayed:

```
J48 unpruned tree
------------------
Binding = Hardcover: Success (5.0/1.0)
Binding = Paperback: Failure (5.0/1.0)
Binding = Leather: Success (3.0/1.0)
Number of Leaves : 3
Size of the tree : 4
```

Testing the book decision tree

We will test the model with two different test cases. Both use identical code to set up the instance. We use the `getTestInstance` method with test-case-specific values and then use the instance with `classifyInstance` to get results. To get something that is more readable, we generate a string, which is then displayed:

```
Instance testInstance = decisionTree.
        getTestInstance("Leather", "yes", "historical");
```

```
int result = (int) tree.classifyInstance(testInstance);
String results = decisionTree.trainingData.attribute(3).value(result);
System.out.println(
        "Test with: " + testInstance + "  Result: " + results);

testInstance = decisionTree.
        getTestInstance("Paperback", "no", "historical");
result = (int) tree.classifyInstance(testInstance);
results = decisionTree.trainingData.attribute(3).value(result);
System.out.println(
        "Test with: " + testInstance + "  Result: " + results);
```

The result of executing this code is as follows:

```
Test with: Leather,yes,historical Result: Success
Test with: Paperback,no,historical Result: Failure
```

This matches our expectations. This technique is based on the amount of information gain before and after an ordering decision has been made. This can be measured based on the entropy as calculated here:

```
Entropy = -portionPos * log2(portionPos) - portionNeg* log2(portionNeg)
```

In this example, `portionPos` is the portion of the data that is positive and `portionNeg` is the portion of the data that is negative. Based on the books file, we can calculate the entropy for the binding as shown in the following table. The information gain is calculated by subtracting the entropy for binding from *1.0*:

Binding	Portion		Entropy
	Positive	Negative	
Hardcover	0.8	0.2	0.72
Leather	0.66	0.33	0.92
Paperback	0.2	0.8	0.72
Entropy for Binding			0.76
Information Gain			0.24

We can calculate the entropy for the use of color and genre in a similar manner. The information gain for color is *0.05*, and it is *0.15* for the genre. Thus, it makes more sense to use the binding type for the first level of the tree.

The resulting tree from the example consists of two levels, because the C4.5 algorithm determines that the remaining features do not provide any additional information gain.

Information gain can be problematic when a feature that has a large number of values is chosen, such as a customer's credit card number. Using this type of attribute will quickly narrow down the field, but it is too selective to be of much value.

Support vector machines

A **Support Vector Machine** (**SVM**) is a supervised machine learning algorithm used for both classification and regression problems. It is mostly used for classification problems. The approach creates a hyperplane to categorize the training data. A hyperplane can be envisioned as a geometric plane that separates two regions. In a two-dimensional space, it will be a line. In a three-dimensional space, it will be a two-dimensional plane. For higher dimensions, it is harder to conceptualize, but they do exist.

Consider the following figure depicting a distribution of two types of data points. The lines represent possible hyperplanes that separate these points. Part of the SVM process is to find the best hyperplane for the problem dataset. We will elaborate on this figure in the coding example.
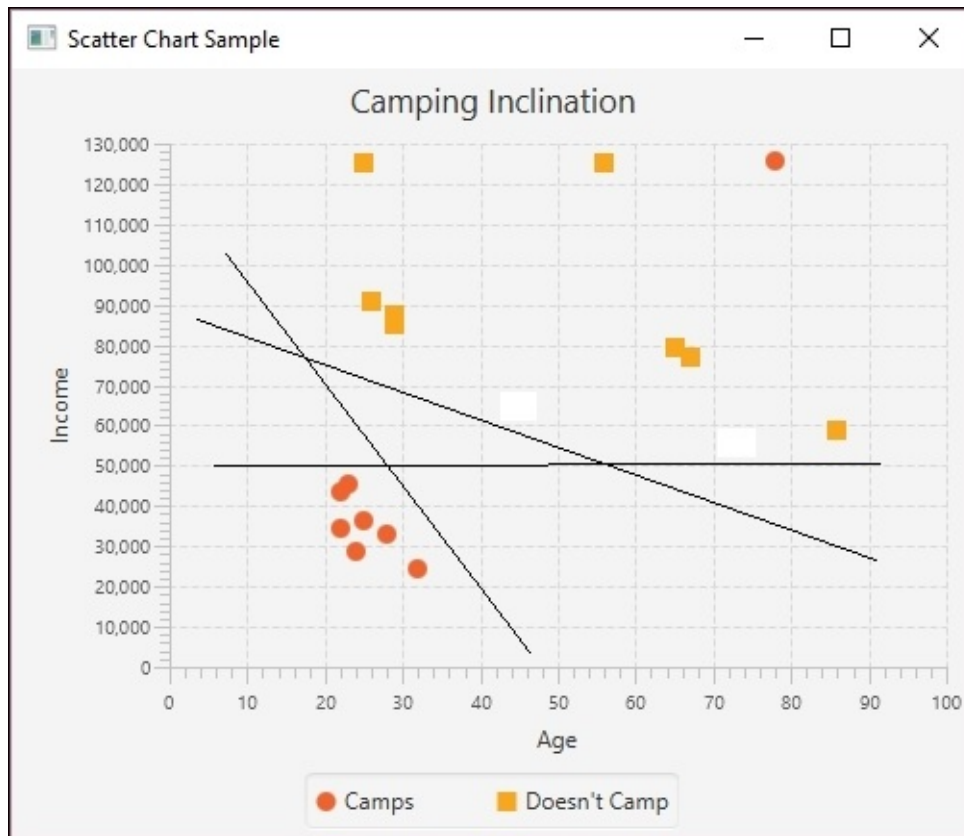
Hyperplane example

Support vectors are data points that lie near the hyperplane. An SVM model uses the concept of a kernel to map input data to a higher order dimensional space to make the data more easily structured. A mapping function for doing this could lead to an infinite-dimensional space; that is, there could be an unbounded number of possible mappings.

However, what is known as the **kernel trick**, a kernel function is an approach that avoids this mapping and avoids possibly infeasible computations that might otherwise occur. SVMs support

different types of kernels. A list of kernels can be found at http://crsouza.com/2010/03/kernel-functions-for-machine-learning-applications/. Choosing the right kernel depends on the problem. Commonly used kernels include:

- **Linear**: Uses a linear hyperplane
- **Polynomial**: Uses a polynomial equation for the hyperplane
- **Radial Basis Function (RBF)**: Uses a non-linear hyperplane
- **Sigmoid**: The sigmoid kernel, also known as the **Hyperbolic Tangent kernel**, comes from the neural networks field and is equivalent to a two-layer perceptron neural network

These kernels support different algorithms for analyzing data.

SVMs are useful for higher dimensional spaces that humans have a harder time visualizing. In the previous figure, two attributes were used to predict a third. An SVM can be used when many more attributes are present. The SVM needs to be trained and this can take longer with larger datasets.

We will use the Weka class `SMO` to demonstrate SVM analysis. The class supports John Platt's sequential minimal optimization algorithm. More information about this algorithm is found at https://www.microsoft.com/en-us/research/publication/fast-training-of-support-vector-machines-using-sequential-minimal-optimization/.

The `SMO` class supports the following kernels, which can be specified when using the class:

- **Puk**: The Pearson VII-function-based universal kernel
- **PolyKernel**: The polynomial kernel
- **RBFKernel**: The RBF kernel

The algorithm uses training data to create a classification model. Test data can then be used to evaluate the model. We can also evaluate individual data elements.

Using an SVM for camping data

For illustration purposes, we will be using a dataset consisting of age, income, and whether someone camps. We would like to be able to predict whether someone is inclined to camp based on their age and income. The data we use is stored in `.arff` format and is not based on a survey but has been created to explain the SVM process. The input data is found in the `camping.txt` file, as shown next. The file extension does not need to be `.arff`:

```
@relation camping
@attribute age numeric
@attribute income numeric
@attribute camps {1, 0}
@data
23,45600,1
45,65700,1
72,55600,1
24,28700,1
22,34200,1
```

```
28,32800,1
32,24600,1
25,36500,1
26,91000,0
29,85300,0
67,76800,0
86,58900,0
56,125300,0
25,125000,0
22,43600,1
78,125700,1
73,56500,1
29,87600,0
65,79300,0
```

The following shows how the data is distributed graphically. Notice the outlier found in the upper-right corner. The JavaFX code that produces this graph is found at http://www.packtpub.com/support:
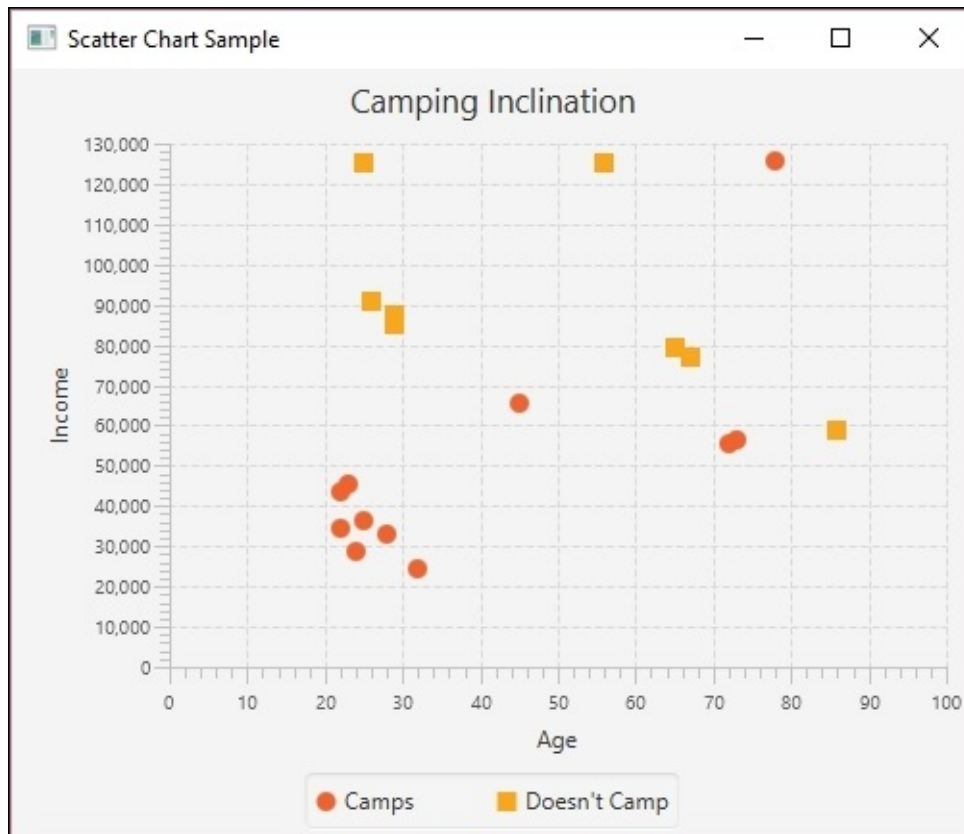
Camping Graph

We will start by reading in the data and handling exceptions:

```
try {
    BufferedReader datafile;
    datafile = readDataFile("camping.txt");
    ...
```

```
} catch (Exception ex) {
    // Handle exceptions
}
```

The `readDataFile` method follows:

```
public BufferedReader readDataFile(String filename) {
    BufferedReader inputReader = null;
    try {
        inputReader = new BufferedReader(
            new FileReader(filename));
    } catch (FileNotFoundException ex) {
        // Handle exceptions
    }
    return inputReader;
}
```

The `Instances` class holds a series of data instances, where each instance is an age, income, and camping value. The `setClassIndex` method indicates which of the attributes is to be predicted. In this case, it is the `camps` attribute:

```
Instances data = new Instances(datafile);
data.setClassIndex(data.numAttributes() - 1);
```

To train the model, we will split the dataset into two sets. The first 14 instances are used to train the model and the last 5 instances are used to test the model. The second argument of the `Instances` constructor specifies the beginning index in the dataset, and the last argument specifies how many instances to include:

```
Instances trainingData = new Instances(data, 0, 14);
Instances testingData = new Instances(data, 14, 5);
```

An `Evaluation` class instance is created to evaluate the model. An instance of the `SMO` class is also created. The `SMO` class's `buildClassifier` method builds the classifier using the dataset:

```
Evaluation evaluation = new Evaluation(trainingData);
Classifier smo = new SMO();
smo.buildClassifier(data);
```

The `evaluateModel` method evaluates the model using the testing data. The results are then displayed:

```
evaluation.evaluateModel(smo, testingData);
System.out.println(evaluation.toSummaryString());
```

The output follows. Notice one incorrectly classified instance. This corresponds to the outlier mentioned earlier:

```
Correctly Classified Instances 4 80 %
Incorrectly Classified Instances 1 20 %
Kappa statistic 0.6154
Mean absolute error 0.2
Root mean squared error 0.4472
```

```
Relative absolute error 41.0256 %
Root relative squared error 91.0208 %
Coverage of cases (0.95 level) 80 %
Mean rel. region size (0.95 level) 50 %
Total Number of Instances 5
```

Testing individual instances

We can also test an individual instance using the `classifyInstance` method. In the following sequence, we create a new instance using the `DenseInstance` class. It is then populated using the attributes of the camping dataset:

```
Instance instance = new DenseInstance(3);
instance.setValue(data.attribute("age"), 78);
instance.setValue(data.attribute("income"), 125700);
instance.setValue(data.attribute("camps"), 1);
```

The instance needs to be associated with the dataset using the `setDataset` method:

```
instance.setDataset(data);
```

The `classifyInstance` method is then applied to the `smo` instance and the results are displayed:

```
System.out.println(smo.classifyInstance(instance));
```

When executed, we get the following output:

```
1.0
```

There are also alternate testing approaches. A common one is called **cross-validation folds**. This approach divides the dataset into *folds,* which are partitions of the dataset. Frequently, 10 partitions are created. Nine of the partitions are used for training and one for testing. This is repeated 10 times using a different partition of the dataset each time, and the average of the results is used. This technique is described at https://weka.wikispaces.com/Generating+cross-validation+folds+(Java+approach).

We will now examine the purpose and use of Bayesian networks.

Bayesian networks

Bayesian networks, also known as **Bayes nets** or **belief networks**, are models designed to reflect a particular world or environment by depicting the states of different attributes of the world and their statistical relationships. The models can be used to show a wide variety of real-world scenarios. In the following diagram, we model a system depicting the relationship between various factors and our likelihood of being late to work:
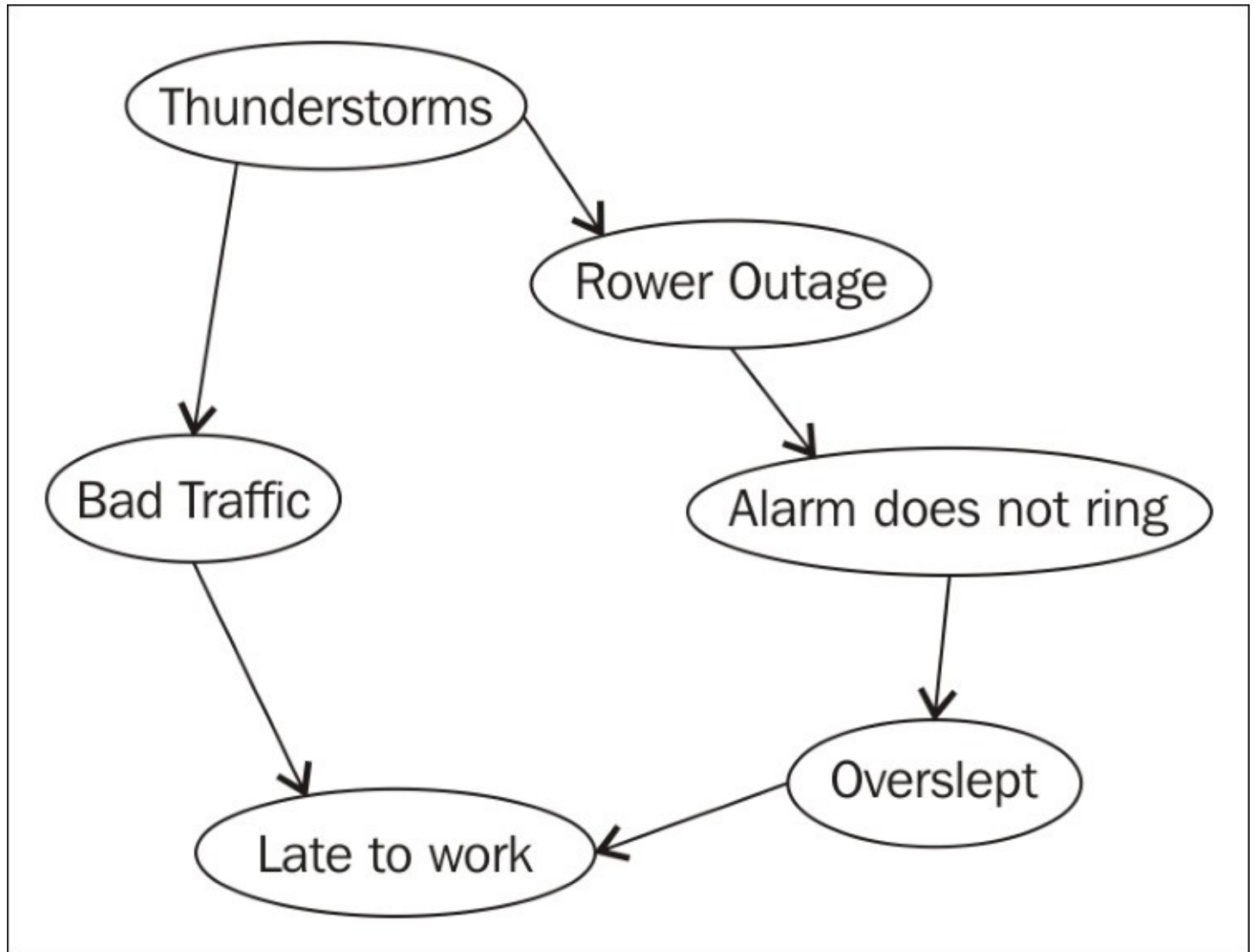
Bayesian network

Each circle on the diagram represents a node or part of the system, which can have various values and probabilities for each value. For example, **Power Outage** might be true or false — either the power went out or it did not. The probability of the power going out affects the probability that your alarm will not ring, you might oversleep, and thus be late to work.

The nodes at the top of the diagram tend to imply a higher level of causality than those at the

bottom. The higher nodes are called **parent nodes**, and they may have one or more child nodes. Bayesian networks only relate nodes that have a causal dependency and therefore allow more efficient computation of probabilities. Unlike other models, we would not have to store and analyze every possible combination of states of each node. Instead, we can calculate and store probabilities of related nodes. Additionally, Bayesian networks are easily adaptable and can grow as more knowledge about a particular world is acquired.

Using a Bayesian network

To model this type of network using Java, we will create a network using JBayes (https://github.com/vangj/jbayes). JBayes is an open source library for creating a simple **Bayesian Belief Network** (**BBN**). It is available at no cost for personal or commercial use. In our next example, we will perform approximate inference, a technique considered less accurate but allowing for decreased computational time. This technique is often used when working with big data because it produces a reliable model in a reasonable amount of time. We conduct approximate inference by using weight sampling of each node. JBayes also provides support for exact inference. Exact inference is most often used with smaller datasets or in situations where accuracy is of most importance. JBayes performs exact inference using the junction tree algorithm.

To begin our approximate inference model, we will first create our nodes. We will use the preceding diagram depicting attributes affecting on-time arrival to work to build our network. In the following code example, we use method chaining to create our nodes. Three of the methods take a `String` parameter. The `name` method is the name associated with each node. For brevity, we are only using the first initial, so *s* represents `storms`, *t* represents `traffic`, and so on. The `value` method allows us to set values for the node. In each case, our nodes can only have two values: `t` for true or `f` for false:

```
Node storms = Node.newBuilder().name("s").value("t").value("f").build();
Node traffic = Node.newBuilder().name("t").value("t").value("f").build();
Node powerOut = Node.newBuilder().name("p").value("t").value("f").build();
Node alarm = Node.newBuilder().name("a").value("t").value("f").build();
Node overslept = Node.newBuilder().name("o").value("t").value("f").build();
Node lateToWork = Node.newBuilder().name("l").value("t").value("f").build();
```

Next, we assign parents to each of our child nodes. Notice that `storms` is a parent node to both `traffic` and `powerOut`. The `lateToWork` node has two parent nodes, `traffic` and `overslept`:

```
traffic.addParent(storms);
powerOut.addParent(storms);
lateToWork.addParent(traffic);
alarm.addParent(powerOut);
overslept.addParent(alarm);
lateToWork.addParent(overslept);
```

Then we define the **conditional probability tables** (**CPTs**) for each of our nodes. These tables are basically two-dimensional arrays representing the probabilities of each attribute of each node. If

we have more than one parent node, as in the case of the `lateToWork` node, we need a row for each. We have used arbitrary values for our probabilities in this example, but note that each row must sum to `1.0`:

```
storms.setCpt(new double[][] {{0.7, 0.3}});
traffic.setCpt(new double[][] {{0.8, 0.2}});
powerOut.setCpt(new double[][] {{0.5, 0.5}});
alarm.setCpt(new double[][] {{0.7, 0.3}});
overslept.setCpt(new double[][] {{0.5, 0.5}});
lateToWork.setCpt(new double[][] {
   {0.5, 0.5},
   {0.5, 0.5}
});
```

Finally, we create a `Graph` object and add each node to our graph structure. We then use this graph to perform our sampling:

```
Graph bayesGraph = new Graph();
bayesGraph.addNode(storms);
bayesGraph.addNode(traffic);
bayesGraph.addNode(powerOut);
bayesGraph.addNode(alarm);
bayesGraph.addNode(overslept);
bayesGraph.addNode(lateToWork);
bayesGraph.sample(1000);
```

At this point, we may be interested in the probabilities of each event. We can use the `prob` method to check the probabilities of a `True` or `False` value for each node:

```
double[] stormProb = storms.probs();
double[] trafProb = traffic.probs();
double[] powerProb = powerOut.probs();
double[] alarmProb = alarm.probs();
double[] overProb = overslept.probs();
double[] lateProb = lateToWork.probs();

out.println("nStorm Probabilities");
out.println("True: " + stormProb[0] + " False: " + stormProb[1]);
out.println("nTraffic Probabilities");
out.println("True: " + trafProb[0] + " False: " + trafProb[1]);
out.println("nPower Outage Probabilities");
out.println("True: " + powerProb[0] + " False: " + powerProb[1]);
out.println("vAlarm Probabilities");
out.println("True: " + alarmProb[0] + " False: " + alarmProb[1]);
out.println("nOverslept Probabilities");
out.println("True: " + overProb[0] + " False: " + overProb[1]);
out.println("nLate to Work Probabilities");
out.println("True: " + lateProb[0] + " False: " + lateProb[1]);
```

Our output contains the probabilities of each value for each node. For example, the probability of a storm occurring is 71% while the probability of a storm not occurring is 29%:

Storm Probabilities

```
True: 0.71 False: 0.29
Traffic Probabilities
True: 0.726 False: 0.274
Power Outage Probabilities
True: 0.442 False: 0.558
Alarm Probabilities
True: 0.543 False: 0.457
Overslept Probabilities
True: 0.556 False: 0.444
Late to Work Probabilities
True: 0.469 False: 0.531
```

Note

Notice in this example that we have used numbers that produce a very high likelihood of being late to work, roughly 47%. This is due to the fact that we have set the probabilities of our parent nodes fairly high as well. This data would vary dramatically if the chances of a storm were lower or if we changed some of the other child nodes as well.

If we would like to save information about our sample, we can save the data to a CSV file using the following code:

```
try {
   CsvUtil.saveSamples(bayesGraph, new FileWriter(
      new File("C://JBayesInfo.csv")));
} catch (IOException e) {
   // Handle exceptions
}
```

With this discussion of supervised learning complete, we will now move on to unsupervised learning.

Unsupervised machine learning

Unsupervised machine learning does not use annotated data; that is, the dataset does to contain anticipated results. While there are several unsupervised learning algorithms, we will demonstrate the use of association rule learning to illustrate this learning approach.

Association rule learning

Association rule learning is a technique that identifies relationships between data items. It is part of what is called **market basket analysis**. When a shopper makes purchases, these purchases are likely to consist of more than one item, and when it does, there are certain items that tend to be bought together. Association rule learning is one approach for identifying these related items. When an association is found, a rule can be formulated for it.

For example, if a customer buys diapers and lotion, they are also likely to buy baby wipes. An analysis can find these associations and a rule stating the observation can be formed. The rule would be expressed as *{diapers, lotion} => {wipes}*. Being able to identify these purchasing patterns allows a store to offer special coupons, arrange their products to be easier to get, or effect any number of other market-related activities.

One of the problems with this technique is that there are a large number of possible associations. One efficient method that is commonly used is the **apriori** algorithm. This algorithm works on a collection of transactions defined by a set of items. These items can be thought of as purchases and a transaction as a set of items bought together. The collection is often referred to as a database.

Consider the following set of transactions where a *1* indicates that the item was purchased as part of a transaction and *0* means that it was not purchased:

Transaction ID	Diapers	Lotion	Wipes	Formula
1	1	1	1	0
2	1	1	1	1
3	0	1	1	0
4	1	0	0	0
5	0	1	1	1

There are several analysis terms used with the apriori model:

- **Support**: This is the proportion of the items in a database that contain a subset of items. In the previous database, the item *{diapers, lotion}* occurs *2/5* times or *20%*.
- **Confidence**: This is a measure of the frequency of the rule being true. It is calculated as *conf(X- > Y) = sup(X ∪ Y)/sup(X)*.

- **Lift**: This measures the degree to which the items are dependent upon each other. It is defined as lift(X -> Y) = *sup(X ∪ Y) / (sup(X) * sup(Y))*.
- **Leverage**: Leverage is a measure of the number of transactions that are covered by both *X* and *Y* if *X* and *Y* are independent of each other. A value above *0* is a good indicator. It is calculated as *lev(X->Y) = sup(X ,Y) - sup(X) * sup(Y)*.
- **Conviction**: A measure of how often the rule makes an incorrect decision. It is defined as *conv(X -> Y) = 1 - sup(Y) / (1 - conf(X -> Y))*.

These definitions and sample values can be found at
https://en.wikipedia.org/wiki/Association_rule_learning.

Using association rule learning to find buying relationships

We will be using the `Apriori` Weka class to demonstrate Java support for the algorithm using two datasets. The first is the data discussed previously and the second deals with what a person may take on a hike.

The following is the data file, `babies.arff`, for baby information:

```
@relation TEST_ITEM_TRANS
@attribute Diapers {1, 0}
@attribute Lotion {1, 0}
@attribute Wipes {1, 0}
@attribute Formula {1, 0}
@data
1,1,1,0
1,1,1,1
0,1,1,0
1,0,0,0
0,1,1,1
```

We start by reading in the file using a `BufferedReader` instance. This object is used as the argument of the `Instances` class, which will hold the data:

```
try {
    BufferedReader br;
    br = new BufferedReader(new FileReader("babies.arff"));
  Instances data = new Instances(br);
    br.close();
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

Next, an `Apriori` instance is created. We set the number of rules to be generated and a minimum confidence for the rules:

```
Apriori apriori = new Apriori();
apriori.setNumRules(100);
apriori.setMinMetric(0.5);
```

The `buildAssociations` method generates the associations using the `Instances` variable. The associations are then displayed:

```
apriori.buildAssociations(data);
System.out.println(apriori);
```

There will be 100 rules displayed. The following is the abbreviated output. Each rule is followed by various measures of the rule:

Note

Note that rule 8 and 100 reflect the previous examples.

```
Apriori
=======
Minimum support: 0.3 (1 instances)
Minimum metric <confidence>: 0.5
Number of cycles performed: 14
Generated sets of large itemsets:
Size of set of large itemsets L(1): 8
Size of set of large itemsets L(2): 18
Size of set of large itemsets L(3): 16
Size of set of large itemsets L(4): 5
Best rules found:
1. Wipes=1 4 ==> Lotion=1 4 <conf:(1)> lift:(1.25) lev:(0.16) [0] conv:(0.8)
2. Lotion=1 4 ==> Wipes=1 4 <conf:(1)> lift:(1.25) lev:(0.16) [0] conv:(0.8)
3. Diapers=0 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0] conv:
(0.4)
4. Diapers=0 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0] conv:(0.4)
5. Formula=1 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0] conv:
(0.4)
6. Formula=1 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0] conv:(0.4)
7. Diapers=1 Wipes=1 2 ==> Lotion=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
8. Diapers=1 Lotion=1 2 ==> Wipes=1 2 <conf:(1)> lift:(1.25) lev:(0.08) [0]
conv:(0.4)
...
62. Diapers=0 Lotion=1 Formula=1 1 ==> Wipes=1 1 <conf:(1)> lift:(1.25) lev:
(0.04) [0] conv:(0.2)
...
99. Lotion=1 Formula=1 2 ==> Diapers=1 1 <conf:(0.5)> lift:(0.83) lev:(-0.04)
[0] conv:(0.4)
100. Diapers=1 Lotion=1 2 ==> Formula=1 1 <conf:(0.5)> lift:(1.25) lev:(0.04)
[0] conv:(0.6)
```

This provides us with a list of relationships, which we can use to identify patterns in activities such as purchasing behavior.

Reinforcement learning

Reinforcement learning is a type of learning at the cutting edge of current research into neural networks and machine learning. Unlike unsupervised and supervised learning, reinforcement learning makes decisions based upon the results of an action. It is a goal-oriented learning process, similar to that used by many parents and teachers across the world. We teach children to study and perform well on tests so that they receive high grades as a reward. Likewise, reinforcement learning can be used to teach machines to make choices that will result in the highest reward.

There are four main components to reinforcement learning: the actor or agent, the state or scenario, the chosen action, and the reward. The actor is the object or vehicle making the decisions within the application. The state is the world the actor exists within. Any decision the actor makes occurs within the parameters of the state. The action is simply the choice the actor makes when given a set of options. The reward is the result of each action and influences the likelihood of choosing that particular action in the future.

It is essential to note that the action and the state where the action occurred are not independent. In fact, the correct, or highest rewarding, action can often depend upon the state in which the action occurs. If the actor is trying to decide how to cross a body of water, swimming might be a good option if the body of water is calm and rather small. Swimming would be a terrible choice for the actor to choose if he wanted to cross the Pacific Ocean.

To handle this problem, we can consider the **Q** function. This function results from the mapping of a particular state to an action within the state. The Q function would link a lower reward to swimming across the Pacific that it might for swimming across a small river. Rather than saying swimming is a low reward activity, the Q function allows for swimming to sometimes have a low reward and other times a higher reward.

Reinforcement learning always begins with a blank slate. The actor does not know the best path or sequence of decisions when the iteration first begins. However, after many iterations through a given problem, considering the results of each particular state-action pair choice, the algorithm improves and learns to make the highest rewarding choices.

The algorithms used to achieve reinforcement learning involve maximization of reward amidst a complex set of processes and choices. Though currently being tested in video games and other discrete environments, the ultimate goal is success of these algorithms in unpredictable, real-world scenarios. Within the topic of reinforcement learning, there are three main flavors or types: temporal difference learning, Q`-learning, and **State-Action-Reward-State-Action (SARSA)**.

Temporal difference learning takes into account previously learned information to inform future decisions. This type of learning assumes a correlation between past and future decisions. Before an action is taken, a prediction is made. This prediction is compared to other known information about the environment and similar decisions before an action is chosen. This process is known as

bootstrapping and is thought to create more accurate and useful results.

Q-learning uses the Q function mentioned above to select not only the best action for one particular step in a given state, but the action that will lead to the highest reward *from that point forward*. This is known as the optimum policy. One great advantage Q-learning offers is the ability to make decisions without requiring a full model of the state. This allows it to function in states with random changes in actions and rewards.

SARSA is another algorithm used in reinforcement learning. Its name is fairly self-explanatory: the Q value depends upon the current **State**, the current chosen **Action**, the **Reward** for that action, the State the agent will exist in after the action is completed, and the subsequent Action taken in the new state. This algorithm looks further ahead one step to make the best possible decision.

There are limited tools currently available for performing reinforcement learning using Java. One popular tool is **Platform for Implementing Q-Learning Experiments** (**Piqle**). This Java framework aims to provide tools for fast designing and testing or reinforcement learning experiments. Piqle can be downloaded from http://piqle.sourceforge.net. Another robust tool is called **Brown-UMBC Reinforcement Learning and Planning** (**BURPLAP**). Found at http://burlap.cs.brown.edu, this library is also designed for development of algorithms and domains for reinforcement learning. This particular resource boasts in the flexibility of states and actions and supports a wide range of planning and learning algorithms. BURLAP also includes analysis tools for visualization purposes.

Summary

Machine learning is concerned with developing techniques that allow the applications to learn without having to be explicitly programmed to solve a problem. This flexibility allows such applications to be used in more varied settings with little to no modifications.

We saw how training data is used to create a model. Once the model has been trained, the model is evaluated using testing data. Both the training data and testing data come from the problem domain. Once trained, the model is used with other input data to make predictions.

We learned how the Weka Java API is used to create decision trees. This tree consists of internal nodes that represent different attributes of the problem. The leaves of the tree represent results. Since there are many ways of constructing a tree, part of the job of a decision tree is to create the best tree.

Support vector machines divide a dataset into sections thus classifying elements in the dataset. This classification s based on the attributes of the data such as age, hair color, or weight. With the model, it is possible to predict outcomes based on attributes of a data instance.

Bayesian networks are used to make predictions based on parent-child relationships between nodes. The probability of one event directly affects the probability of a child event, and we can use this information to predict outcomes of complex real-world environments.

In the section on association rule learning, we learned how the relationships between elements of a dataset can be identified. The more significant relationships allow us to create rules that are applied to solve various problems.

In our discussion of reinforcement learning, we discussed the elements of agent, state, action, and reward and their relationship to one another. We also discussed specific types of reinforcement learning and provided resources for further inquiry.

Having introduced the elements of machine learning, we are now ready to explore neural networks, found in the next chapter.

Chapter 7. Neural Networks

While neural networks have been around for a number of years, they have grown in popularity due to improved algorithms and more powerful machines. Some companies are building hardware systems that explicitly mimic neural networks (https://www.wired.com/2016/05/google-tpu-custom-chips/). The time has come to use this versatile technology to address data science problems.

In this chapter, we will explore the ideas and concepts behind neural networks and then demonstrate their use. Specifically, we will:

- Define and illustrate neural networks
- Describe how they are trained
- Examine various neural network architectures
- Discuss and demonstrate several different neural networks, including:
  - A simple Java example
  - A **Multi Layer Perceptron** (**MLP**) network
  - The **k-Nearest Neighbor** (**k-NN**) algorithm and others

The idea for an **Artificial Neural Network** (**ANN**), which we will call a neural network, originates from the neuron found in the brain. A **neuron** is a cell that has **dendrites** connecting it to input sources and other neurons. It receives stimulus from multiple sources through the **dendrites**. Depending on the source, the weight allocated to a source, the neuron is activated and **fires** a signal down a dendrite to another neuron. A collection of neurons can be trained and will respond to a particular set of input signals.

An artificial neuron is a node that has one or more inputs and a single output. Each input has a weight associated with it. By weighting inputs, we can amplify or de-amplify an input.

Note

Artificial neurons are alternately called **perceptrons**.

This is depicted in the following diagram, where the weights are summed and then sent to an **Activation Function** that determines the **Output.**
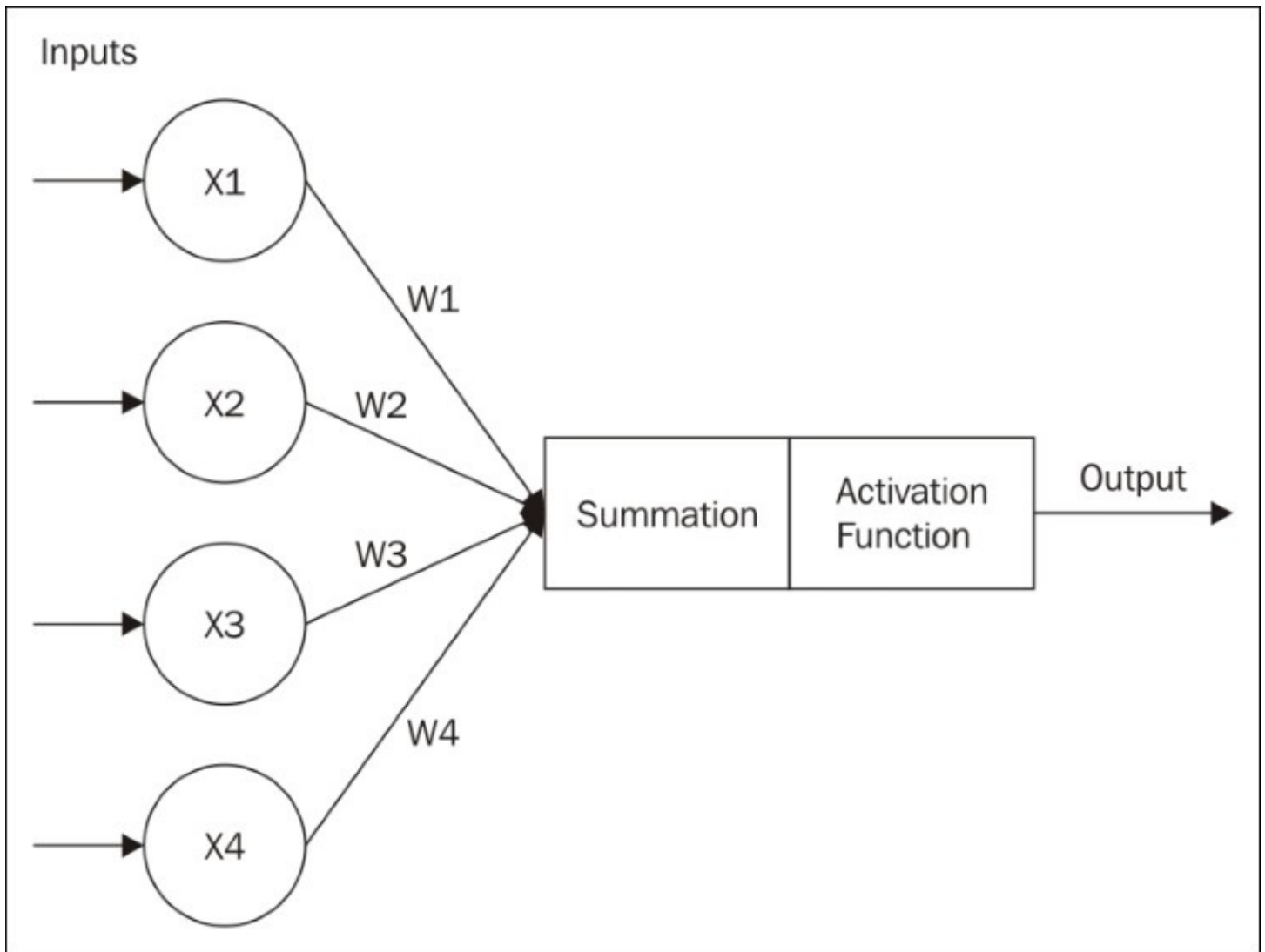
The neuron, and ultimately a collection of neurons, operate in one of two modes:

- **Training mode** - The neuron is trained to fire when a certain set of inputs are received
- **Testing mode** - Input is provided to the neuron, which responds as trained to a known set of inputs

A dataset is frequently split into two parts. A larger part is used to train a model. The second part is used to test and verify the model.

The output of a neuron is determined by the sum of the weighted inputs. Whether a neuron fires or not is determined by an **activation function**. There are several different types of activations functions, including:

- **Step function** - This linear function is computed using the summation of the weighted inputs as follows:

$$Net_i = \sum W_i X_i$$

The *f(Net)* designates the output of a function. It is *1*, if the *Net* input is greater than the activation threshold. When this happens the neuron fires. Otherwise it returns *0* and doesn't fire. The value is calculated based on all of the dendrite inputs.

- **Sigmoid** - This is a nonlinear function and is computed as follows:

$$f(Net) = \frac{1}{1 + e^{-Net_i}}$$

As the neuron is trained, the weights with each input can be adjusted.

In contrast to the step function, the sigmoid function is non-linear. This better matches some problem domains. We will find the sigmoid function used in multi-layer neural networks.

Training a neural network

There are three basic training approaches:

- **Supervised learning** - With supervised learning the model is trained with data that matches input sets to output values
- **Unsupervised learning** - In unsupervised learning, the data does not contain results, but the model is expected to determine relationships on its own
- **Reinforcement learning** - Similar to supervised learning, but a reward is provided for good results

These datasets differ in the information they contain. Supervised and reinforcement learning contain correct output for a set of input. The unsupervised learning does not contain correct results.

A neural network learns (at least with supervised learning) by feeding an input into a network and comparing the results, using the activation function, to the expected outcome. If they match, then the network has been trained correctly. If they don't match then the network is modified.

When we modify the weights we need to be careful not to change them too drastically. If the change is too large, then the results may change too much and we may miss the desired output. If the change is too little, then training the model will take too long. There are times when we may not want to change some weights.

A **bias** unit is a neuron that has a constant output. It is always one and is sometimes referred to as a fake node. This neuron is similar to an offset and is essential for most networks to function properly. You could compare the bias neuron to the y-intercept of a linear function in slope-intercept form. Just as adjusting the y-intercept value changes the location of the line, but not the shape/slope, the bias neuron can change the output values without adjusting the shape or function of the network. You can adjust the outputs to fit the particular needs of your problem.

Getting started with neural network architectures

Neural networks are usually created using a series of layers of neurons. There is typically an **Input Layer**, one or more middle layers (**Hidden Layer**), and an **Output Layer**.

The following is the depiction of a feedforward network:
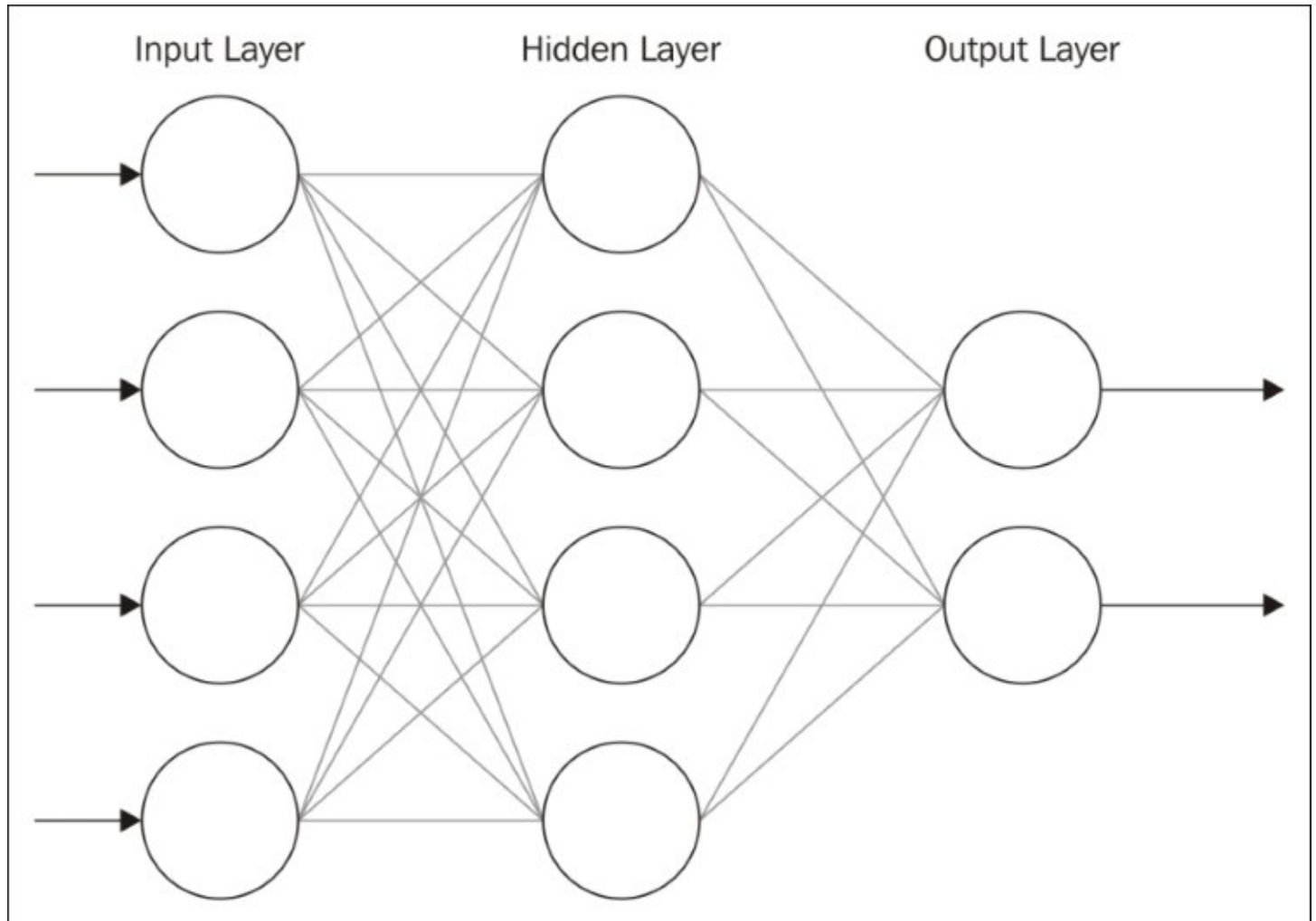
The number of nodes and layers will vary. A feedforward network moves the information forward. There are also feedback networks where information is passed backwards. Multiple hidden layers are needed to handle the more complicated processing required for most analysis.

We will discuss several architectures and algorithms related to different types of neural networks throughout this chapter. Due to the complexity and length of explanation required, we will only provide in-depth analysis of a few key network types. Specifically, we will demonstrate a simple neural network, MLPs, and **Self-Organizing Maps** (**SOMs**).

We will, however, provide an overview of many different options. The type of neural network and algorithm implementation appropriate for any particular model will depend upon the problem

being addressed.

Understanding static neural networks

Static neural networks are ANNs that undergo a training or learning phase and then do not change when they are used. They differ from dynamic neural networks, which learn constantly and may undergo structural changes after the initial training period. Static neural networks are useful when the results of a model are relatively easy to reproduce or are more predictable. We will look at dynamic neural networks in a moment, but we will begin by creating our own basic static neural network.

A basic Java example

Before we examine various libraries and tools available for constructing neural networks, we will implement our own basic neural network using standard Java libraries. The next example is an adaptation of work done by Jeff Heaton (http://www.informit.com/articles/article.aspx?p=30596). We will construct a feed-forward backpropagation neural network and train it to recognize the XOR operator pattern. Here is the basic truth table for XOR:

X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

This network needs only two input neurons and one output neuron corresponding to the *X* and *Y* input and the result. The number of input and output neurons needed for models is dependent upon the problem at hand. The number of hidden neurons is often the sum of the number of input and output neurons, but the exact number may need to be changed as training progresses.

We are going to demonstrate how to create and train the network next. We first provide the network with an input and observe the output. The output is compared to the expected output and then the weight matrix, called `weightChanges`, is adjusted. This adjustment ensures that the subsequent output will be closer to the expected output. This process is repeated until we are satisfied that the network can produce results significantly close enough to the expected output. In this example, we present the input and output as arrays of doubles where each input or output neuron is an element of the array.

Note

The input and output are sometimes referred to as **patterns**.

First, we will create a `SampleNeuralNetwork` class to implement the network. Begin by adding the variables listed underneath to the class. We will discuss and demonstrate their purposes later in this section. Our class contains the following instance variables:

```
double errors;
int inputNeurons;
int outputNeurons;
int hiddenNeurons;
```

```
    int totalNeurons;
    int weights;
    double learningRate;
    double outputResults[];
    double resultsMatrix[];
    double lastErrors[];
    double changes[];
    double thresholds[];
    double weightChanges[];
    double allThresholds[];
    double threshChanges[];
    double momentum;
    double errorChanges[];
```

Next, let's take a look at our constructor. We have four parameters, representing the number of inputs to our network, the number of neurons in hidden layers, the number of output neurons, and the rate and momentum at which we wish for learning to occur. The `learningRate` is a parameter that specifies the magnitude of changes in weight and bias during the training process. The `momentum` parameter specifies what fraction of a previous weight should be added to create a new weight. It is useful to prevent convergence at **local minimums** or **saddle points**. A high momentum increases the speed of convergence in a system, but can lead to an unstable system if it is too high. Both the momentum and learning rate should be values between `0` and `1`:

```
public SampleNeuralNetwork(int inputCount,
        int hiddenCount,
        int outputCount,
        double learnRate,
        double momentum) {
    ...
}
```

Within our constructor we initialize all private instance variables. Notice that `totalNeurons` is set to the sum of all inputs, outputs, and hidden neurons. This sum is then used to set several other variables. Also notice that the `weights` variable is calculated by finding the product of the number of inputs and hidden neurons, the product of the hidden neurons and the outputs, and adding these two products together. This is then used to create new arrays of length weight:

```
    learningRate = learnRate;
    momentum = momentum;

    inputNeurons = inputCount;
    hiddenNeurons = hiddenCount;
    outputNeurons = outputCount;
    totalNeurons = inputCount + hiddenCount + outputCount;
    weights = (inputCount * hiddenCount)
        + (hiddenCount * outputCount);

    outputResults  = new double[totalNeurons];
    resultsMatrix  = new double[weights];
    weightChanges = new double[weights];
    thresholds = new double[totalNeurons];
    errorChanges = new double[totalNeurons];
```

```
        lastErrors    = new double[totalNeurons];
        allThresholds = new double[totalNeurons];
        changes = new double[weights];
        threshChanges = new double[totalNeurons];
        reset();
```

Notice that we call the `reset` method at the end of the constructor. This method resets the network to begin training with a random weight matrix. It initializes the thresholds and results matrices to random values. It also ensures that all matrices used for tracking changes are set back to zero. Using random values ensures that different results can be obtained:

```
public void reset() {
    int loc;
    for (loc = 0; loc < totalNeurons; loc++) {
        thresholds[loc] = 0.5 - (Math.random());
        threshChanges[loc] = 0;
        allThresholds[loc] = 0;
    }
    for (loc = 0; loc < resultsMatrix.length; loc++) {
        resultsMatrix[loc] = 0.5 - (Math.random());
        weightChanges[loc] = 0;
        changes[loc] = 0;
    }
}
```

We also need a method called `calcThreshold`. The **threshold** value specifies how close a value has to be to the actual activation threshold before the neuron will fire. For example, a neuron may have an activation threshold of `1`. The threshold value specifies whether a number such as `0.999` counts as `1`. This method will be used in subsequent methods to calculate the thresholds for individual values:

```
public double threshold(double sum) {
    return 1.0 / (1 + Math.exp(-1.0 * sum));
}
```

Next, we will add a method to calculate the output using a given set of inputs. Both our input parameter and the data returned by the method are arrays of `double` values. First, we need two position variables to use in our loops, `loc` and `pos`. We also want to keep track of our position within arrays based upon the number of input and hidden neurons. The index for our hidden neurons will start after our input neurons, so its position is the same as the number of input neurons. The position of our output neurons is the sum of our input neurons and hidden neurons. We also need to initialize our `outputResults` array:

```
public double[] calcOutput(double input[]) {
    int loc, pos;
    final int hiddenIndex = inputNeurons;
    final int outIndex = inputNeurons + hiddenNeurons;

    for (loc = 0; loc < inputNeurons; loc++) {
        outputResults[loc] = input[loc];
```

```
    }
...
}
```

Then we calculate outputs based upon our input neurons for the first layer of our network. Notice our use of the `threshold` method within this section. Before we can place our sum in the `outputResults` array, we need to utilize the `threshold` method:

```
int rLoc = 0;
for (loc = hiddenIndex; loc < outIndex; loc++) {
    double sum = thresholds[loc];
    for (pos = 0; pos < inputNeurons; pos++) {
        sum += outputResults[pos] * resultsMatrix[rLoc++];
    }
    outputResults[loc] = threshold(sum);
}
```

Now we take into account our hidden neurons. Notice the process is similar to the previous section, but we are calculating outputs for the hidden layer rather than the input layer. At the end, we return our result. This result is in the form of an array of doubles containing the values of each output neuron. In our example, there is only one output neuron:

```
double result[] = new double[outputNeurons];
for (loc = outIndex; loc < totalNeurons; loc++) {
    double sum = thresholds[loc];

    for (pos = hiddenIndex; pos < outIndex; pos++) {
        sum += outputResults[pos] * resultsMatrix[rLoc++];
    }
    outputResults[loc] = threshold(sum);
    result[loc-outIndex] = outputResults[loc];
}

return result;
```

It is quite likely that the output does not match the expected output, given our XOR table. To handle this, we use error calculation methods to adjust the weights of our network to produce better output. The first method we will discuss is the `calcError` method. This method will be called every time a set of outputs is returned by the `calcOutput` method. It does not return data, but rather modifies arrays containing weight and threshold values. The method takes an array of doubles representing the ideal value for each output neuron. Notice we begin as we did in the `calcOutput` method and set up indexes to use throughout the method. Then we clear out any existing hidden layer errors:

```
public void calcError(double ideal[]) {
    int loc, pos;
    final int hiddenIndex = inputNeurons;
    final int outputIndex = inputNeurons + hiddenNeurons;

    for (loc = inputNeurons; loc < totalNeurons; loc++) {
```

```
            lastErrors[loc] = 0;
    }
```

Next we calculate the difference between our expected output and our actual output. This allows us to determine how to adjust the weights for further training. To do this, we loop through our arrays containing the expected outputs, `ideal`, and the actual outputs, `outputResults`. We also adjust our errors and change in errors in this section:

```
    for (loc = outputIndex; loc < totalNeurons; loc++) {
        lastErrors[loc] = ideal[loc - outputIndex] -
            outputResults[loc];
        errors += lastErrors[loc] * lastErrors[loc];
        errorChanges[loc] = lastErrors[loc] * outputResults[loc]
            *(1 - outputResults[loc]);
    }

    int locx = inputNeurons * hiddenNeurons;
    for (loc = outputIndex; loc < totalNeurons; loc++) {
        for (pos = hiddenIndex; pos < outputIndex; pos++) {
            changes[locx] += errorChanges[loc] *
                outputResults[pos];
            lastErrors[pos] += resultsMatrix[locx] *
                errorChanges[loc];
            locx++;
        }
        allThresholds[loc] += errorChanges[loc];
    }
```

Next we calculate and store the change in errors for each neuron. We use the `lastErrors` array to modify the `errorChanges` array, which contains total errors:

```
for (loc = hiddenIndex; loc < outputIndex; loc++) {
    errorChanges[loc] = lastErrors[loc] *outputResults[loc]
        * (1 - outputResults[loc]);
}
```

We also fine tune our system by making changes to the `allThresholds` array. It is important to monitor the changes in errors and thresholds so the network can improve its ability to produce correct output:

```
    locx = 0;
    for (loc = hiddenIndex; loc < outputIndex; loc++) {
        for (pos = 0; pos < hiddenIndex; pos++) {
            changes[locx] += errorChanges[loc] *
                outputResults[pos];
            lastErrors[pos] += resultsMatrix[locx] *
                errorChanges[loc];
            locx++;
        }
        allThresholds[loc] += errorChanges[loc];
```

```
      }
}
```

We have one other method used for calculating errors in our network. The `getError` method calculates the root mean square for our entire set of training data. This allows us to identify our average error rate for the data:

```
public double getError(int len) {
    double err = Math.sqrt(errors / (len * outputNeurons));
    errors = 0;
    return err;
}
```

Now that we can initialize our network, compute outputs, and calculate errors, we are ready to train our network. We accomplish this through the use of the `train` method. This method makes adjustments first to the weights based upon the errors calculated in the previous method, and then adjusts the thresholds:

```
public void train() {
    int loc;
    for (loc = 0; loc < resultsMatrix.length; loc++) {
        weightChanges[loc] = (learningRate * changes[loc]) +
            (momentum * weightChanges[loc]);
        resultsMatrix[loc] += weightChanges[loc];
        changes[loc] = 0;
    }
    for (loc = inputNeurons; loc < totalNeurons; loc++) {
        threshChanges[loc] = learningRate * allThresholds[loc] +
            (momentum * threshChanges[loc]);
        thresholds[loc] += threshChanges[loc];
        allThresholds[loc] = 0;
    }
}
```

Finally, we can create a new class to test our neural network. Within the `main` method of another class, add the following code to represent the XOR problem:

```
double xorIN[][] ={
                {0.0,0.0},
                {1.0,0.0},
                {0.0,1.0},
                {1.0,1.0}};

double xorEXPECTED[][] = { {0.0},{1.0},{1.0},{0.0}};
```

Next we want to create our new `SampleNeuralNetwork` object. In the following example, we have two input neurons, three hidden neurons, one output neuron (the XOR result), a learn rate of `0.7`, and a momentum of `0.9`. The number of hidden neurons is often best determined by trial and error. In subsequent executions, consider adjusting the values in this constructor and examine the difference in results:

```
SampleNeuralNetwork network = new
```

```
SampleNeuralNetwork(2,3,1,0.7,0.9);
```

Note

The learning rate and momentum should usually fall between zero and one.

We then repeatedly call our `calcOutput`, `calcError`, and `train` methods, in that order. This allows us to test our output, calculate the error rate, adjust our network weights, and then try again. Our network should display increasingly accurate results:

```
for (int runCnt=0;runCnt<10000;runCnt++) {
   for (int loc=0;loc<xorIN.length;loc++) {
        network.calcOutput(xorIN[loc]);
        network.calcError(xorEXPECTED[loc]);
        network.train();
   }
   System.out.println("Trial #" + runCnt + ",Error:" +
              network.getError(xorIN.length));
}
```

Execute the application and notice that the error rate changes with each iteration of the loop. The acceptable error rate will depend upon the particular network and its purpose. The following is some sample output from the preceding code. For brevity we have included the first and the last training output. Notice that the error rate is initially above 50%, but falls to close to 1% by the last run:

```
Trial #0,Error:0.5338334002845255
Trial #1,Error:0.5233475199946769
Trial #2,Error:0.5229843653785426
Trial #3,Error:0.5226263062497853
Trial #4,Error:0.5226916275713371
...
Trial #994,Error:0.014457034704806316
Trial #995,Error:0.01444865096401158
Trial #996,Error:0.01444028142777395
Trial #997,Error:0.014431926056394229
Trial #998,Error:0.0144235848103274
Trial #999,Error:0.014415257650182488
```

In this example, we have used a small scale problem and we were able to train our network rather quickly. In a larger scale problem, we would start with a training set of data and then use additional datasets for further analysis. Because we really only have four inputs in this scenario, we will not test it with any additional data.

This example demonstrates some of the inner workings of a neural network, including details about how errors and output can be calculated. By exploring a relatively simple problem we are able to examine the mechanics of a neural network. In our next examples, however, we will use tools that hide these details from us, but allow us to conduct robust analysis.

Understanding dynamic neural networks

Dynamic neural networks differ from static networks in that they continue learning after the training phase. They can make adjustments to their structure independently of external modification. A **feedforward neural network (FNN)** is one of the earliest and simplest dynamic neural networks. This type of network, as its name implies, only feeds information forward and does not form any cycles. This type of network formed the foundation for much of the later work in dynamic ANNs. We will show in-depth examples of two types of dynamic networks in this section, MLP networks and SOMs.

Multilayer perceptron networks

A MLP network is a FNN with multiple layers. The network uses supervised learning with backpropagation where feedback is sent to early layers to assist in the learning process. Some of the neurons use a nonlinear activation function mimicking biological neurons. Every nodes of one layer is fully connected to the following layer.

We will use a dataset called `dermatology.arff` that can be downloaded from http://repository.seasr.org/Datasets/UCI/arff/. This dataset contains 366 instances used to diagnosis erythemato-squamous diseases. It uses 34 attributes to classify the disease into one of five different categories. The following is a sample instance:

`2,2,0,3,0,0,0,0,1,0,0,0,0,0,0,3,2,0,0,0,0,0,0,0,0,0,0,3,0,0,0,1,0,55,2`

The last field represents the disease category. This dataset has been partitioned into two files: `dermatologyTrainingSet.arff` and `dermatologyTestingSet.arff`. The training set uses the first 80% (292 instances) of the original set and ends with line 456. The testing set is the last 20% (74 instances) and starts with line 457 of the original set (lines 457-530).

Building the model

Before we can make any predictions, it is necessary that we train the model on a representative set of data. We will use the Weka class, `MultilayerPerceptron`, for training and eventually to make predictions. First, we declare strings for the training and testing of filenames and the corresponding `FileReader` instances for them. The instances are created and the last field is specified as the field to use for classification:

```
String trainingFileName = "dermatologyTrainingSet.arff";
String testingFileName = "dermatologyTestingSet.arff";

try (FileReader trainingReader = new FileReader(trainingFileName);
        FileReader testingReader =
            new FileReader(testingFileName)) {
    Instances trainingInstances = new Instances(trainingReader);
    trainingInstances.setClassIndex(
        trainingInstances.numAttributes() - 1);
    Instances testingInstances = new Instances(testingReader);
    testingInstances.setClassIndex(
        testingInstances.numAttributes() - 1);
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

An instance of the `MultilayerPerceptron` class is then created:

```
MultilayerPerceptron mlp = new MultilayerPerceptron();
```

There are several model parameters that we can set, as shown here:

Parameter	Method	Description
Learning rate	`setLearningRate`	Affects the training speed
Momentum	`setMomentum`	Affects the training speed
Training time	`setTrainingTime`	The number of training epochs used to train the model
Hidden layers	`setHiddenLayers`	The number of hidden layers and perceptrons to use

As mentioned previously, the learning rate will affect the speed in which your model is trained. A large value can increase the training speed. If the learning rate is too small, then the training time may take too long. If the learning rate is too large, then the model may move past a local minimum and become divergent. That is, if the increase is too large, we might skip over a meaningful value. You can think of this a graph where a small dip in a plot along the *Y* axis is missed because we incremented our *X* value too much.

Momentum also affects the training speed by effectively increasing the rate of learning. It is used in addition to the learning rate to add momentum to the search for the optimal value. In the case of a local minimum, the momentum helps get out of the minimum in its quest for a global minimum.

When the model is learning it performs operations iteratively. The term, **epoch** is used to refer to the number of iterations. Hopefully, the total error encounter with each epoch will decrease to a point where further epochs are not useful. It is ideal to avoid too many epochs.

A neural network will have one or more hidden layers. Each of these layers will have a specific number of perceptrons. The `setHiddenLayers` method specifies the number of layers and perceptrons using a string. For example, *3,5* would specify two hidden layers with three and five perceptrons per layer, respectively.

For this example, we will use the following values:

```
mlp.setLearningRate(0.1);
mlp.setMomentum(0.2);
mlp.setTrainingTime(2000);
mlp.setHiddenLayers("3");
```

The `buildClassifier` method uses the training data to build the model:

```
mlp.buildClassifier(trainingInstances);
```

Evaluating the model

The next step is to evaluate the model. The `Evaluation` class is used for this purpose. Its

constructor takes the training set as input and the `evaluateModel` method performs the actual evaluation. The following code illustrates this using the testing dataset:

```
Evaluation evaluation = new Evaluation(trainingInstances);
evaluation.evaluateModel(mlp, testingInstances);
```

One simple way of displaying the results of the evaluation is using the `toSummaryString` method:

```
System.out.println(evaluation.toSummaryString());
```

This will display the following output:

```
Correctly Classified Instances 73 98.6486 %
Incorrectly Classified Instances 1 1.3514 %
Kappa statistic 0.9824
Mean absolute error 0.0177
Root mean squared error 0.076
Relative absolute error 6.6173 %
Root relative squared error 20.7173 %
Coverage of cases (0.95 level) 98.6486 %
Mean rel. region size (0.95 level) 18.018 %
Total Number of Instances 74
```

Frequently, it will be necessary to experiment with these parameters to get the best results. The following are the results of varying the number of perceptrons:

Number of perceptrons	Correctly classified instances		Incorrectly classified instances	
	Number	Percentage	Number	Percentage
2	55	74.3243%	19	25.6757%
3	73	98.6486%	1	1.3514%
4	72	97.2973%	2	2.7027%
5	72	97.2973%	2	2.7027%

Predicting other values

Once we have a model trained, we can use it to evaluate other data. In the previous testing dataset there was one instance which failed. In the following code sequence, this instance is identified and the predicted and actual results are displayed.

Each instance of the testing dataset is used as input to the `classifyInstance` method. This method tries to predict the correct result. This result is compared to the last field of the instance

that contains the actual value. For mismatches, the predicted and actual values are displayed:

```
for (int i = 0; i < testingInstances.numInstances(); i++) {
    double result = mlp.classifyInstance(
        testingInstances.instance(i));
    if (result != testingInstances
            .instance(i)
            .value(testingInstances.numAttributes() - 1)) {
        out.println("Classify result: " + result
                + " Correct: " + testingInstances.instance(i)
                .value(testingInstances.numAttributes() - 1));
        ...
    }
}
```

For the testing set we get the following output:

Classify result: 1.0 Correct: 3.0

We can get the likelihood of the prediction being correct using the `MultilayerPerceptron` class' `distributionForInstance` method. Place the following code into the previous loop. It will capture the incorrect instance, which is easier than instantiating an instance based on the 34 attributes used by the dataset. The `distributionForInstance` method takes this instance and returns a two element array of doubles. The first element is the probability of the result being positive and the second is the probability of it being negative:

```
Instance incorrectInstance = testingInstances.instance(i);
incorrectInstance.setDataset(trainingInstances);
double[] distribution = mlp.distributionForInstance(incorrectInstance);
out.println("Probability of being positive: " + distribution[0]);
out.println("Probability of being negative: " + distribution[1]);
```

The output for this instance is as follows:

Probability of being positive: 0.00350515156929017
Probability of being negative: 0.9683660500711128

This can provide a more quantitative feel for the reliability of the prediction.

Saving and retrieving the model

We can also save and retrieve a model for later use. To save the model, build the model and then use the `SerializationHelper` class' static method `write`, as shown in the following code snippet. The first argument is the name of the file to hold the model:

```
SerializationHelper.write("mlpModel", mlp);
```

To retrieve the model, use the corresponding `read` method as shown here:

```
mlp = (MultilayerPerceptron)SerializationHelper.read("mlpModel");
```

Next, we will learn how to use another useful neural network approach, SOMs.

Learning vector quantization

Learning Vector Quantization (**LVQ**) is another special type of a dynamic ANN. SOMs, which we will discuss in a moment, are a by-product of LVQ networks. This type of network implements a competitive type of algorithm in which the winning neuron gains the weight. These types of networks are used in many different applications and are considered to be more natural and intuitive than some other ANNs. In particular, LVQ is effective for classification of text-based data.

The basic algorithm begins by setting the number of neurons, the weight for each neuron, how fast the neurons can learn, and a list of input vectors. In this context, a vector is similar to a vector in physics and represents the values provided to the input layer neurons. As the network is trained, a vector is used as input, a winning neuron is selected, and the weight of the winning neuron is updated. This model is iterative and will continue to run until a solution is found.

Self-Organizing Maps

SOMs is a technique that takes multidimensional data and reducing it to one or two dimensions. This compression technique is called **vector quantization**. The technique usually involves a visual component that allows a human to better see how the data has been categorized. SOM learns without supervision.

The SOM is good for finding clusters, which is not to be confused with classification. With classification we are interested in finding the best fit for a data instance among predefined categories. With clustering we are interested in grouping instances where the categories are unknown.

A SOM uses a lattice of neurons, usually a two-dimensional array or a hexagonal grid, representing neurons that are assigned weights. The input sources are connected to each of these neurons. The technique then adjusts the weights assigned to each lattice member through several iterations until the best fit is found. When finished, the lattice members will have grouped the input dataset into categories. The SOM results can be viewed to identify categories and map new input to one of the identified categories.

Using a SOM

We will use the Weka to demonstrate SOM. However, it is does not come installed with standard Weka. Instead, we will need to download a set of Weka classification algorithms from https://sourceforge.net/projects/wekaclassalgos/files/ and the actual SOM class from http://www.cis.hut.fi/research/som_pak/. The classification algorithms include support for LVQ. More details about the classification algorithms can be found at http://wekaclassalgos.sourceforge.net/.

To use the SOM class, called `SelfOrganizingMap`, the source code needs to be in your project. The Javadoc for this class is found at http://jsalatas.ictpro.gr/weka/doc/SelfOrganizingMap/.

We start with the creation of an instance of the `SelfOrganizingMap` class. This is followed by code to read in data and create an `Instances` object to hold the data. In this example, we will use the `iris.arff` file, which can be found in the Weka data directory. Notice that once the `Instances` object is created we do not specify the class index as we did with previous Weka examples since SOM uses unsupervised learning:

```
SelfOrganizingMap som = new SelfOrganizingMap();
String trainingFileName = "iris.arff";
try (FileReader trainingReader =
        new FileReader(trainingFileName)) {
    Instances trainingInstances = new Instances(trainingReader);
    ...
} catch (IOException ex) {
    // Handle exceptions
} catch (Exception ex) {
    // Handle exceptions
```

```
}
```

The `buildClusterer` method will execute the SOM algorithm using the training dataset:

```
som.buildClusterer(trainingInstances);
```

Displaying the SOM results

We can now display the results of the operation as follows:

```
out.println(som.toString());
```

The `iris` dataset uses five attributes: `sepallength`, `sepalwidth`, `petallength`, `petalwidth`, and `class`. The first four attributes are numeric and the fifth has three possible values: `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`. The first part of the abbreviated output that follows identified four clusters and the number of instances in each cluster. This is followed by statistics for each of the attributes:

```
Self Organized Map
==================
Number of clusters: 4
Cluster
Attribute 0 1 2 3
(50) (42) (29) (29)
===========================================
sepallength
value 5.0036 6.2365 5.5823 6.9513
min 4.3 5.6 4.9 6.2
max 5.8 7 6.3 7.9
mean 5.006 6.25 5.5828 6.9586
std. dev. 0.3525 0.3536 0.3675 0.5046
...
class
value 0 1.5048 1.0787 2
min 0 1 1 2
max 0 2 2 2
mean 0 1.4524 1.069 2
std. dev. 0 0.5038 0.2579 0
```

These statistics can provide insight into the dataset. If we are interested in determining which dataset instance is found in a cluster, we can use the `getClusterInstances` method to return the array that groups the instances by cluster. As shown next, this method is used to list the instance by cluster:

```
Instances[] clusters = som.getClusterInstances();
int index = 0;
for (Instances instances : clusters) {
    out.println("-------Custer " + index);
    for (Instance instance : instances) {
        out.println(instance);
    }
    out.println();
    index++;
```

}

As we can see with the abbreviated output of this sequence, different `iris` classes are grouped into the different clusters:

```
-------Custer 0
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
...
5.3,3.7,1.5,0.2,Iris-setosa
5,3.3,1.4,0.2,Iris-setosa
-------Custer 1
7,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
...
6.5,3,5.2,2,Iris-virginica
5.9,3,5.1,1.8,Iris-virginica

-------Custer 2
5.5,2.3,4,1.3,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
4.9,2.4,3.3,1,Iris-versicolor
...
4.9,2.5,4.5,1.7,Iris-virginica
6,2.2,5,1.5,Iris-virginica

-------Custer 3
6.3,3.3,6,2.5,Iris-virginica
7.1,3,5.9,2.1,Iris-virginica
6.5,3,5.8,2.2,Iris-virginica
...
```

The cluster results can be displayed visually using the Weka GUI interface. In the following screenshot, we have used the **Weka Workbench** to analyze and visualize the result of the SOM analysis:
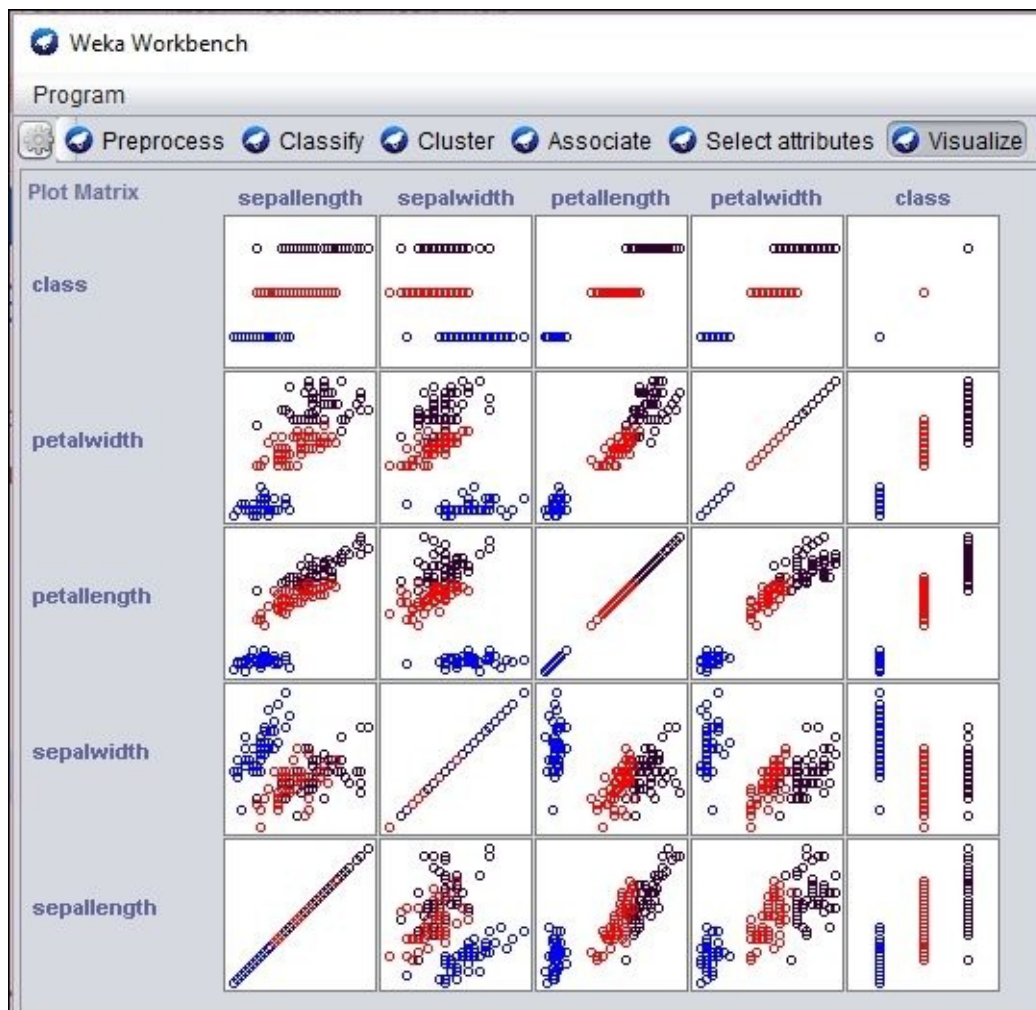
An individual section of the graph can be selected, customized, and analyzed as follows:

However, before you can use the `SOM` class, the `WekaPackageManager` must be used to add the `SOM` package to Weka. This process is discussed at https://weka.wikispaces.com/How+do+I+use+the+package+manager%3F.

If a new instance needs to be mapped to a cluster, the `distributionForInstance` method can be used as illustrated in *Predicting other values* section.

Additional network architectures and algorithms

We have discussed a few of the most common and practical neural networks. At this point, we would also like to consider some specialized neural networks and their application in various fields of study. These types of networks do not fit neatly into one particular category, but may still be of interest.

The k-Nearest Neighbors algorithm

An artificial neural network implementing the k-NN algorithm is similar to MLP networks, but it provides significant reduction in time compared to the winner takes all strategy. This type of network does not require a training algorithm after the initial weights are set and has fewer connections among its neurons. We have chosen not to provide an example of this algorithm's implementation because its use in Weka is very similar to the MLP example.

This type of network is best suited to classification tasks. Because it utilizes lazy learning techniques, reserving all computation until after information has been classified, it is considered to be one of the simpler models. In this model, the neurons are weighted based upon their distance from their neighbors. The classification of the neighbors is already known and therefore no specific training is required.

Instantaneously trained networks

Instantaneously Trained Neural Networks (**ITNNs**) are feedforward ANNs. They are special because they add a new hidden neuron for every unique set of training data. The main advantage to this type of network is the ability to provide generalization to other problems.

ITNNs are especially useful in short term learning situations. In particular, this type of network is useful for web searches and other pattern recognition functions with large datasets. These networks are suited for time series prediction and other deep learning purposes.

Spiking neural networks

A **Spiking Neural Network** (**SNN**) is a more complex ANN due to the fact it takes into account not only the neuron and information propagation, but also the timing of each event. In these networks, every neuron does not fire during every propagation of information, but rather only when the **membrane potential** for a particular neuron reaches a specific threshold. The membrane potential refers to the activation level of a neuron and closely resembles the way biological neurons fire.

Due to the close mimicry of biological neural networks, SNNs are especially suited to biological study and application. They have been used to model the nervous system of animals and insects and are useful for predicting the outcome of various stimuli. These networks have the ability to create very complex models with significant detail, but sacrifice time to accomplish this goal.

Cascading neural networks

Cascading Neural Networks (**CNNs**) is a specialized supervised learning algorithm. In this type, the network is initially very small and simplistic. As the network learns, it gradually adds new hidden units. Once the node is added, its input weight is constant and cannot be changed or removed.

This type of neural network is praised for its quick learning rate and ability to dynamically build itself. The user of such a network does not have to worry about topological design. Additionally, these networks do not require backpropagation of error information to make adjustments.

Holographic associative memory

Holographic Associative Memory (**HAM**) is a special type of complex neural network. This is a specialized type of network related to natural human memory and visual analysis. This network is especially useful for pattern recognition and associative memory tasks and can be applied to optical computations.

HAM attempts to closely mimic human visualization and pattern recognition. In this network, stimulus-response patterns are learned without iteration and backpropagation of errors is not required. Unlike other networks discussed in this chapter, HAM does not exhibit the same type of connected behaviour. Instead, the stimulus-response patterns can be stored within a single neuron.

Backpropagation and neural networks

Backpropagation algorithms are another supervised learning techniques used to train neural networks. As the name suggests, this algorithm calculates the computed output error and then changes the weights of each neuron in a backwards manner. Backpropagation is primarily used with MLP networks. It is important to note forward propagation must occur before backward propagation can be used.

In its most basic form, this algorithm consists of four steps:

1. Perform forward propagation for a given set of inputs.
2. Calculate the error value for each output.
3. Change the weights based upon the calculated error for each node.
4. Perform forward propagation again.

This algorithm completes when the output matches the expected output.

Summary

In this chapter, we have provided a broad overview of artificial neural networks, as well as a detailed examination of a few specific implementations. We began with a discussion of the basic properties of neural networks, training algorithms, and neural network architectures.

Next we provided an example of a simple static neural network implementing the XOR problem using Java. This example provided detailed explanation of the code used to build and train the network, including some of the math behind the weight adjustments during the training process. We then discussed dynamic neural networks and provided two in-depth examples, the MLP and SOM networks. These used the Weka tools to create and train the networks.

Finally, we concluded our chapter with a discussion of additional network architectures and algorithms. We chose some of the more popular networks to summarize and explored situations where each type would be most useful. We also included a discussion of backpropagation in this section.

In the next chapter, we will expand upon this introduction and take a look at deep learning with neural networks.

Chapter 8. Deep Learning

In this chapter, we will focus on neural networks, often referred to as **Deep Learning Networks** (**DLNs**). This type of network is characterized as a multiple-layer neural network. Each of these layers are rained on the output of the previous layer, potentially identifying features and sub-features of the dataset. A feature hierarchy is created in this manner.

DLNs typically work with unstructured and unlabeled data, which constitute the vast bulk of data found in the world today. DLN will take this unstructured data, identify features, and try to reconstruct the original input. This approach is illustrated with **Restricted Boltzmann Machines** (**RBMs**) in *Restricted Boltzmann Machines* and with autoencoders in *Deep autoencoders*. An autoencoder takes a dataset and effectively compresses it. It then decompresses it to reconstruct the original dataset.

DLN can also be used for predictive analysis. The last step of a DLN will use an activation function to generate output represented by one of several categories. When used with new data, the model will attempt to classify the input based on the previously trained model.

An important DLN task is ensuring that the model is accurate and minimizes error. As with simple neural networks, weights and biases are used at each layer. As weight values are adjusted, errors can be introduced. A technique to adjust weights uses **gradient descent**. This can be thought of as the slope of the change. The idea is to modify the weight so as to minimize the error. It is an optimization technique that speeds up the learning process.

Later in the chapter, we will examine **Convolutional Neural Networks** (**CNNs**) and briefly discuss **Recurrent Neural Networks** (**RNN**). Convolution networks mimic the visual cortex in that each neuron can interact with and make decisions **based** on a region of information. Recurrent networks process information based on not only the output of the previous layer but also the calculations performed in previous layers.

There are several libraries that support deep learning, including these:

- **N-Dimensional Arrays for Java** (**ND4J**) http://nd4j.org/: A scientific computing library intended for production use
- **Deeplearning4j** (http://deeplearnin4j.org/): An open source, distributed deep-learning library
- **Encog** (http://www.heatonresearch.com/encog/): This library supports several deep learning algorithms

ND4J is a lower level library that is actually used in other projects, including DL4J. Encog is perhaps not as well supported as DL4J, but does provide support for deep learning.

The examples used in this chapter are all based on the **Deep Learning for Java** (**DL4J**) (http://deeplearning4j.org) API with support from ND4J. This library provides good support for many of the algorithms associated with deep learning. As a result, the next section explains the

basic tasks found in common with many of the deep learning algorithms, such as loading data, training a model, and testing the model.

Deeplearning4j architecture

In this section, we will discuss its architecture and address several of the common tasks performed when using the API. DLN typically starts with the creation of a `MultiLayerConfiguration` instance, which defines the network, or model. The network is composed of multiple layers. **Hyperparameters** are used to configure the network and are variables that affect such things as learning speed, activation functions to use for a layer, and how weights are to be initialized.

As with neural networks, the basic DLN process consists of:

- Acquiring and manipulating data
- Configuring and building a model
- Training the model
- Testing the model

We will investigate each of these tasks in the next sections.

Note

The code examples in this section are not intended to be entered and executed here. Instead, these examples are snippets out of later models that we will be using.

Acquiring and manipulating data

The DL4J API has a number of techniques for acquiring data. We will focus on those specific techniques that we will use in our examples. The dataset used by a DL4J project is often modified using either **binarization** or **normalization**. Binarization converts data to ones and zeroes. Normalization converts data to a value between *1* and *0*.

Data feed to DLN is transformed to a set of numbers. These numbers are referred to as **vectors**. These vectors consist of a one-column matrix with a variable number of rows. The process of creating a vector is called **vectorization**.

Canova (http://deeplearning4j.org/canova.html) is a DL4J library that supports vectorization. It works with many different types of datasets. It has been merged with **DataVec** (http://deeplearning4j.org/datavec), a vectorization and **Extract, Transform, and Load** (**ETL**) library.

In this section, we will focus on how to read in CSV data.

Reading in a CSV file

ND4J provides the `CSVRecordReader` class, which is useful for reading CSV data. It has three overloaded constructors. The one we will demonstrate is passed two arguments. The first is the number of lines to skip when first reading a file and the second is a string holding the delimiters used to parse the text.

In the following code, we create a new instance of the class, where we do not skip any lines and use only a comma for a delimiter:

```
RecordReader recordReader = new CSVRecordReader(0, ",");
```

The class implements the `RecordReader` interface. It has an `initialize` method that is passed an instance of the `FileSplit` class. One of its constructors is passed an instance of a `File` object that references a dataset. The `FileSplit` class assists in splitting the data for training and testing. In this example, we initialize the reader for a file called `car.txt` that we will use in the *Preparing the data* section:

```
recordReader.initialize(new FileSplit(new File("car.txt")));
```

To process the data, we need an iterator such as the `DataSetIterator` instance shown next. This class possesses a multitude of overloaded constructors. In the following example, the first argument is the `RecordReader` instance. This is followed by three arguments. The first is the batch size, which is the number of records to retrieve at a time. The next one is the index of the last attribute of the record. The last argument is the number of classes represented by the dataset:

```
DataSetIterator iterator =
    new RecordReaderDataSetIterator(recordReader, 1728, 6, 4);
```

The file's record's last attribute will hold a class value if we use a dataset for regression. This is precisely how we will use it later. The number of the class's parameter is only used with regression.

In the next code sequence, we will split the dataset into two sets: one for training and one for testing. Starting with the `next` method, this method returns the next dataset from the source. The size of the dataset is dependent on the batch size used earlier. The `shuffle` method randomizes the input while the `splitTestAndTrain` method returns an instance of the `SplitTestAndTrain` class, which we use to get the training and testing datasets. The `splitTestAndTrain` method's argument specifies the percentage of the data to be used for training.

```
DataSet dataset = iterator.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

We can then use these datasets with a model.

Configuring and building a model

Frequently, DL4J uses the `MultiLayerConfiguration` class to define the configuration of the model and the `MultiLayerNetwork` class to represent a model. These classes provide a flexible way of building models.

In the following example, we will demonstrate the use of these classes. Starting with the `MultiLayerConfiguration` class, we find that several methods are used in a fluent style. We will provide more details about these methods shortly. However, notice that two layers are defined for this model:

```
MultiLayerConfiguration conf =
    new NeuralNetConfiguration.Builder()
        .iterations(1000)
        .activation("relu")
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.4)
        .list()
        .layer(0, new DenseLayer.Builder()
                .nIn(6).nOut(3)
                .build())
        .layer(1, new OutputLayer
                .Builder(LossFunctions.LossFunction
                        .NEGATIVELOGLIKELIHOOD)
                .activation("softmax")
                .nIn(3).nOut(4).build())
        .backprop(true).pretrain(false)
        .build();
```

The `nIn` and `nOut` methods specify the number of inputs and outputs for a layer.

Using hyperparameters in ND4J

Builder classes are common in DL4J. In the previous example, the `NeuralNetConfiguration.Builder` class is used. The methods used here are but a few of the many that are available. In the following table, we describe several of them:

Method	Usage
iterations	Controls the number of optimization iterations performed
activation	This is the activation function used
weightInit	Used to initialize the initial weights for the model
learningRate	Controls the speed the model learns

`List`	Creates an instance of the `NeuralNetConfiguration.ListBuilder` class so that we can add layers
`Layer`	Creates a new layer
`backprop`	When set to true, it enables backpropagation
`pretrain`	When set to true, it will pretrain the model
`Build`	Performs the actual build process

Let's examine how a layer is created more closely. In the example, the `list` method returns a `NeuralNetConfiguration.ListBuilder` instance. Its `layer` method takes two arguments. The first is the number of the layer, a zero-based numbering scheme. The second is the `Layer` instance.

There are two different layers used here with two different builders: a `DenseLayer.Builder` and an `OutputLayer.Builder` instance. There are several types of layers available in DL4J. The argument of a builder's constructor may be a **loss function**, as is the case with the output layer, and is explained next.

In a feedback network, the neural network's guess is compared to what is called the **ground truth**, which is the error. This error is used to update the network through the modification of weights and biases. The loss function, also called an **objective** or **cost function**, measures the difference.

There are several loss functions supported by DL4J:

- `MSE`: In linear regression MSE stands for mean squared error
- `EXPLL`: In poisson regression EXPLL stands for exponential log likelihood
- `XENT`: In binary classification XENT stands for cross entropy
- `MCXENT`: This stands for multiclass cross entropy
- `RMSE_XENT`: This stands for RMSE cross entropy
- `SQUARED_LOSS`: This stands for squared loss
- `RECONSTRUCTION_CROSSENTROPY`: This stands for reconstruction cross entropy
- `NEGATIVELOGLIKELIHOOD`: This stands for negative log likelihood
- `CUSTOM`: Define your own loss function

The remaining methods used with the builder instance are the activation function, the number of inputs and outputs for the layer, and the `build` method, which creates the layer.

Each layer of a multi-layer network requires the following:

- **Input**: Usually in the form of an input vector

- **Weights**: Also called coefficients
- **Bias**: Used to ensure that at least some nodes in a layer are activated
- **Activation function**: Determines whether a node fires

There are many different types of activation functions, each of which can address a particular type of problem.

The activation function is used to determine whether the neuron fires. There are several functions supported, including `relu` (rectified linear), `tanh`, `sigmoid`, `softmax`, `hardtanh`, `leakyrelu`, `maxout`, `softsign`, and `softplus`.

Note

An interesting list of activation functions along with graphs is found at http://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons and https://en.wikipedia.org/wiki/Activation_function.

Instantiating the network model

Next, a `MultiLayerNetwork` instance is created using the defined configuration. The model is initialized, and its listeners are set. The `ScoreIterationListener` instance will display information as the model trains, which we will see shortly. Its constructor's argument specifies how often that information should be displayed:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(100));
```

We are now ready to train the model.

Training a model

This is actually a fairly simple step. The `fit` method performs the training:

```
model.fit(trainingData);
```

When executed, the output will be generated using any listeners associated with the model, as is the preceding case, where a `ScoreIterationListener` instance is used.

Another example of how the `fit` method is used is through the process of iterating through a dataset, as shown next. In this example, a sequence of datasets is used. This is the part of an autoencoder where the output is intended to match the input, as explained in *Deep autoencoders* section. The dataset used as the argument to the `fit` method uses both the input and the expected output. In this case, they are the same as provided by the `getFeatureMatrix` method:

```
while (iterator.hasNext()) {
    DataSet dataSet = iterator.next();
    model.fit(new DataSet(dataSet.getFeatureMatrix(),
            dataSet.getFeatureMatrix()));
}
```

For larger datasets, it is necessary to pretrain the model several times to get accurate results. This is often performed in parallel to reduce training time. This option is set with a layer class's `pretrain` method.

Testing a model

The evaluation of a model is performed using the `Evaluation` class and the training dataset. An `Evaluation` instance is created using an argument specifying the number of classes. The test data is fed into the model using the `output` method. The `eval` method takes the output of the model and compares it against the test data classes to generate statistics:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
out.println(evaluation.stats());
```

The output will look similar to the following:

```
==========================Scores===================================
Accuracy: 0.9273
Precision: 0.854
Recall: 0.8323
F1 Score: 0.843
```

These statistics are detailed here:

- `Accuracy`: This is a measure of how often the correct answer was returned.
- `Precision`: This is a measure of the probability that a positive response is correct.
- `Recall`: This measures how likely the result will be classified correctly if given a positive example.
- `F1 Score`: This is the probability that the network's results are correct. It is the harmonic mean of recall and precision. It is calculated by dividing the number of true positives by the sum of true positives and false negatives.

We will use the `Evaluation` class to determine the quality of our model. A measure called **f1** is used, whose values range from *0* to *1*, where *1* represents the best quality.

Deep learning and regression analysis

Neural networks can be used to perform regression analysis. However, other techniques (see the early chapters) may offer a more effective solution. With regression analysis, we want to predict a result based on several input variables.

We can perform regression analysis using an output layer that consists of a single neuron that sums the weighted input plus bias of the previous hidden layer. Thus, the result is a single value representing the regression.

Preparing the data

We will use a car evaluation database to demonstrate how to predict the acceptability of a car based on a series of attributes. The file containing the data we will be using can be downloaded from: http://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data. It consists of car data such as price, number of passengers, and safety information, and an assessment of its overall quality. It is this latter element, quality, that we will try to predict. The comma-delimited values in each attribute are shown next, along with substitutions. The substitutions are needed because the model expects numeric data:

Attribute	Original value	Substituted value
Buying price	vhigh, high, med, low	3,2,1,0
Maintenance price	vhigh, high, med, low	3,2,1,0
Number of doors	2, 3, 4, 5-more	2,3,4,5
Seating	2, 4, more	2,4,5
Cargo space	small, med, big	0,1,2
Safety	low, med, high	0,1,2

There are 1,728 instances in the file. The cars are marked with four classes:

Class	Number of instances	Percentage of instances	Original value	Substituted value
Unacceptable	1210	70.023%	unacc	0
Acceptable	384	22.222%	acc	1
Good	69	3.99%	good	2
Very good	65	3.76%	v-good	3

Setting up the class

We start with the definition of a `CarRegressionExample` class, as shown next, where an instance of the class is created and where the work is performed within its default constructor:

```
public class CarRegressionExample {

    public CarRegressionExample() {
        try {
            ...
        } catch (IOException | InterruptedException ex) {
            // Handle exceptions
        }
    }

    public static void main(String[] args) {
        new CarRegressionExample();
    }
}
```

Reading and preparing the data

The first task is to read in the data. We will use the `CSVRecordReader` class to get the data, as explained in *Reading in a CSV file*:

```
RecordReader recordReader = new CSVRecordReader(0, ",");
recordReader.initialize(new FileSplit(new File("car.txt")));
DataSetIterator iterator = new
  RecordReaderDataSetIterator(recordReader, 1728, 6, 4);
```

With this dataset, we will split the data into two sets. Sixty five percent of the data is used for training and the rest for testing:

```
DataSet dataset = iterator.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

The data now needs to be normalized:

```
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainingData);
normalizer.transform(trainingData);
normalizer.transform(testData);
```

We are now ready to build the model.

Building the model

A `MultiLayerConfiguration` instance is created using a series of `NeuralNetConfiguration.Builder` methods. The following is the dice used. We will discuss the individual methods following the code. Note that this configuration uses two layers. The last layer uses the `softmax` activation function, which is used for regression analysis:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .iterations(1000)
        .activation("relu")
        .weightInit(WeightInit.XAVIER)
        .learningRate(0.4)
        .list()
        .layer(0, new DenseLayer.Builder()
                .nIn(6).nOut(3)
                .build())
        .layer(1, new OutputLayer
                .Builder(LossFunctions.LossFunction
                        .NEGATIVELOGLIKELIHOOD)
                .activation("softmax")
                .nIn(3).nOut(4).build())
        .backprop(true).pretrain(false)
        .build();
```

Two layers are created. The first is the input layer. The `DenseLayer.Builder` class is used to create this layer. The `DenseLayer` class is a feed-forward and fully connected layer. The created layer uses the six car attributes as input. The output consists of three neurons that are fed into the output layer and is duplicated here for your convenience:

```
.layer(0, new DenseLayer.Builder()
        .nIn(6).nOut(3)
        .build())
```

The second layer is the output layer created with the `OutputLayer.Builder` class. It uses a loss function as the argument of its constructor. The `softmax` activation function is used since we are performing regression as shown here:

```
.layer(1, new OutputLayer
        .Builder(LossFunctions.LossFunction
                .NEGATIVELOGLIKELIHOOD)
        .activation("softmax")
        .nIn(3).nOut(4).build())
```

Next, a `MultiLayerNetwork` instance is created using the configuration. The model is initialized, its listeners are set, and then the `fit` method is invoked to perform the actual training. The `ScoreIterationListener` instance will display information as the model trains which we will see shortly in the output of this example. The `ScoreIterationListener` constructor's argument specifies the frequency that information is displayed:

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

```
model.setListeners(new ScoreIterationListener(100));
model.fit(trainingData);
```

We are now ready to evaluate the model.

Evaluating the model

In the next sequence of code, we evaluate the model against the training dataset. An `Evaluation` instance is created using an argument specifying that there are four classes. The test data is fed into the model using the `output` method. The `eval` method takes the output of the model and compares it against the test data classes to generate statistics. The `getLabels` method returns the expected values:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
out.println(evaluation.stats());
```

The output of the training follows, which is produced by the `ScoreIterationListener` class. However, the values you get may differ due to how the data is selected and analyzed. Notice that the score improves with the iterations but levels out after about 500 iterations:

```
12:43:35.685 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
0 is 1.443480901811554
12:43:36.094 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
100 is 0.3259061845624861
12:43:36.390 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
200 is 0.2630572026049783
12:43:36.676 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
300 is 0.24061281470878784
12:43:36.977 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
400 is 0.22955121170274934
12:43:37.292 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
500 is 0.22249920540161677
12:43:37.575 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
600 is 0.2169898450109222
12:43:37.872 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
700 is 0.21271599814600958
12:43:38.161 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
800 is 0.2075677126088741
12:43:38.451 [main] INFO o.d.o.l.ScoreIterationListener - Score at iteration
900 is 0.20047317735870715
```

This is followed by the results of the `stats` method as shown next. The first part reports on how examples are classified and the second part displays various statistics:

```
Examples labeled as 0 classified by model as 0: 397 times
Examples labeled as 0 classified by model as 1: 10 times
Examples labeled as 0 classified by model as 2: 1 times
Examples labeled as 1 classified by model as 0: 8 times
Examples labeled as 1 classified by model as 1: 113 times
Examples labeled as 1 classified by model as 2: 1 times
Examples labeled as 1 classified by model as 3: 1 times
Examples labeled as 2 classified by model as 1: 7 times
Examples labeled as 2 classified by model as 2: 21 times
Examples labeled as 2 classified by model as 3: 14 times
Examples labeled as 3 classified by model as 1: 2 times
Examples labeled as 3 classified by model as 3: 30 times
```

```
==========================Scores==================================Accuracy:
0.9273
Precision: 0.854
Recall: 0.8323
F1 Score: 0.843
=================================================================
```

The regression model does a reasonable job with this dataset.

Restricted Boltzmann Machines

RBM is often used as part of a multi-layer deep belief network. The output of the RBM is used as an input to another layer. The use of the RBM is repeated until the final layer is reached.

Note

Deep Belief Networks (**DBNs**) consist of several RBMs stacked together. Each hidden layer provides the input for the subsequent layer. Within each layer, the nodes cannot communicate laterally and it becomes essentially a network of other single-layer networks. DBNs are especially helpful for classifying, clustering, and recognizing image data.

The term, **continuous restricted Boltzmann machine**, refers an RBM that uses values other than integers. Input data is normalized to values between zero and one.

Each node of the input layer is connected to each node of the second layer. No nodes of the same layer are connected to each other. That is, there is no intra-layer communication. This is what restricted means.
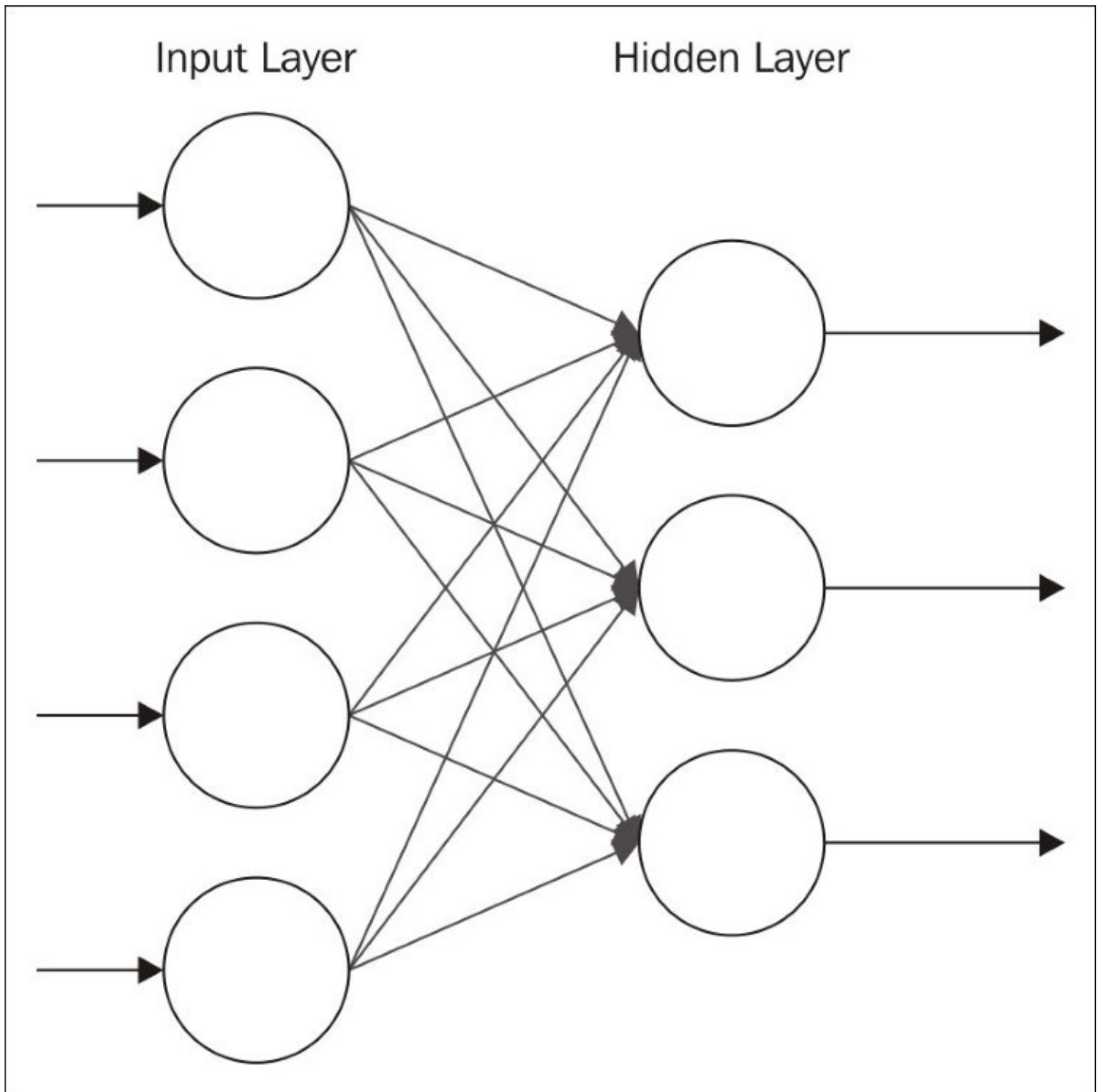
The number of input nodes for the visible layer is dependent on the problem being solved. For example, if we are looking at an image with *256* pixels, then we will need *256* input nodes. For an image, this is the number of rows times the number of columns for the image.

The **Hidden Layer** should contain fewer neurons than the **Input Layer**. Using close to the same number of neurons will sometimes result in the construction of an identity function. Too many neurons may result in overfitting. This means that datasets with a large number of inputs will require multiple layers. Smaller input sizes result in the need for fewer layers.

Stochastic, that is, random, values are assigned to each node's weights. The value for a node is multiplied by its weight and then added to a bias. This value, combined with the combined input from the other input nodes, is then fed into the activation function, where an output value is generated.

Reconstruction in an RBM

The RBM technique goes through a reconstruction phase. This is where the activations are fed back to the first layer and multiplied by the same weights used for the input. The sum of these values from each node of the second layer, plus another bias, represents an approximation of the original input. The idea is to train the model to minimize the difference between the original input values and the feedback values.
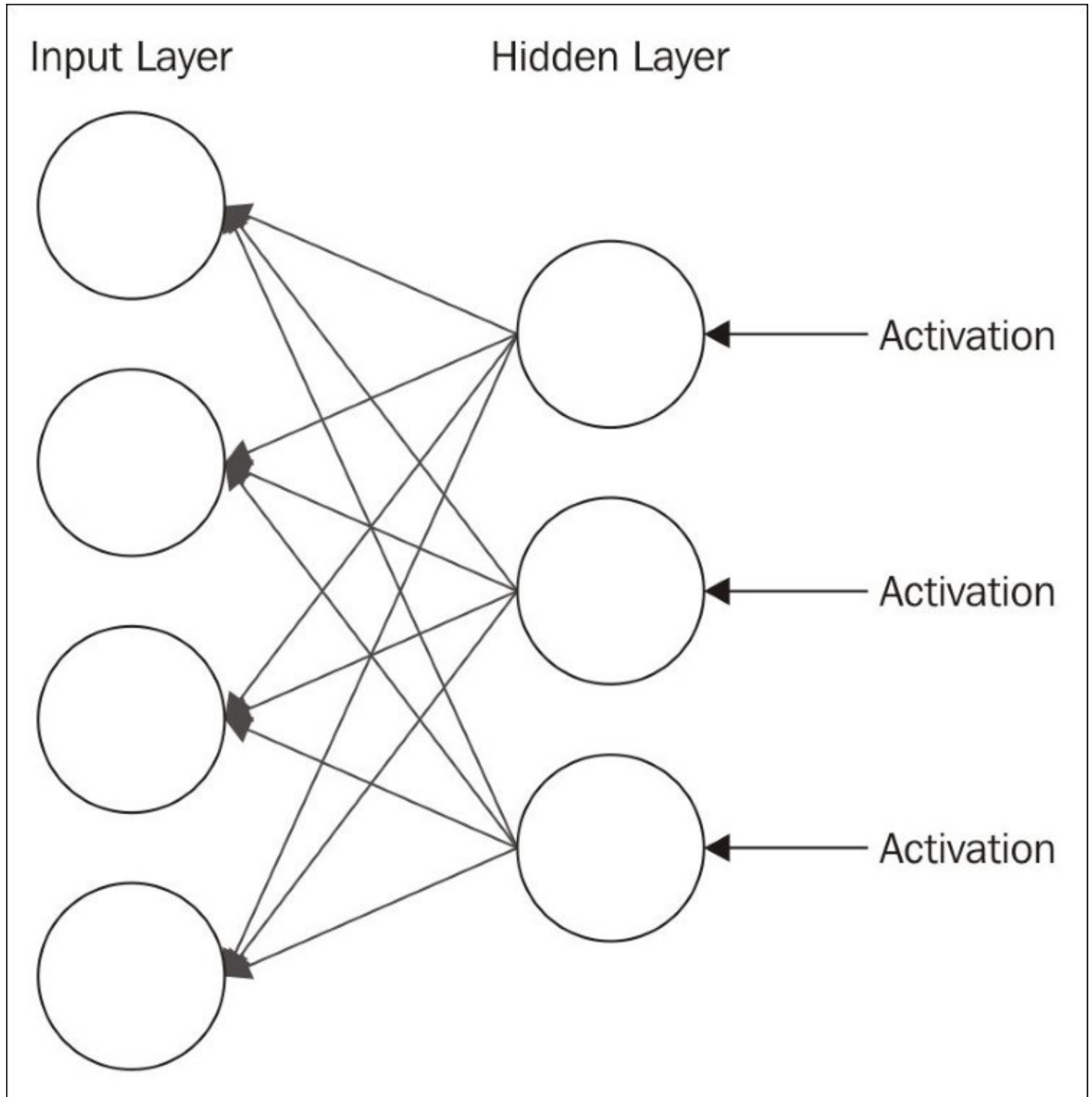
The difference in values is treated as an error. The process is repeated until an error minimum is reached. You can think of the reconstruction as guesses about the original input. These guesses are essentially a probability distribution of the original input. This is called generative learning, in contrast to discriminative learning, which occurs with classification techniques.

In a multi-layer model, each layer can be used to essentially identify a feature. In subsequent layers, a combination of features may be identified or generated. In this way, a seemingly random set of pixel values may be analyzed to identify the veins of a leaf, a leaf, a trunk, and then a tree.

The output of an RBM is a value that essentially represents a percentage. If it is not zero, then the machine has learned something about the input.

Configuring an RBM

We will examine two different RBM configurations. The first one is minimal and we will see it again in *Deep autoencoders*. The second uses several additional methods and provides more insights into the various ways it can be configured.

The following statement creates a new layer using the `RBM.Builder` class. The input is computed based on the number of rows and columns of an image. The output is large, containing `1000` neurons. The loss function is `RMSE_XENT`. This loss function works better for some classification problems:

```
.layer(0, new RBM.Builder()
    .nIn(numRows * numColumns).nOut(1000)
    .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
    .build())
```

Next is a more complex RBM. We will not detail each of these methods here but will see them used in later examples:

```
.layer(new RBM.Builder()
    .l2(1e-1).l1(1e-3)
    .nIn(numRows * numColumns
    .nOut(outputNum)
    .activation("relu")
    .weightInit(WeightInit.RELU)
    .lossFunction(LossFunctions.LossFunction
        .RECONSTRUCTION_CROSSENTROPY).k(3)
    .hiddenUnit(HiddenUnit.RECTIFIED)
    .visibleUnit(VisibleUnit.GAUSSIAN)
    .updater(Updater.ADAGRAD)
        .gradientNormalization(
            GradientNormalization.ClipL2PerLayer)
    .build())
```

A single-layer `RBM` is not always useful. A multi-layer autoencoder is often required. We will look at this option in the next section.

Deep autoencoders

An autoencoder is used for feature selection and extraction. It consists of two symmetrical DBNs. The first half of the network is composed of several layers, which performs encoding. The second part of the network performs decoding. Each layer of the autoencoder is an RBM. This is illustrated in the following figure:
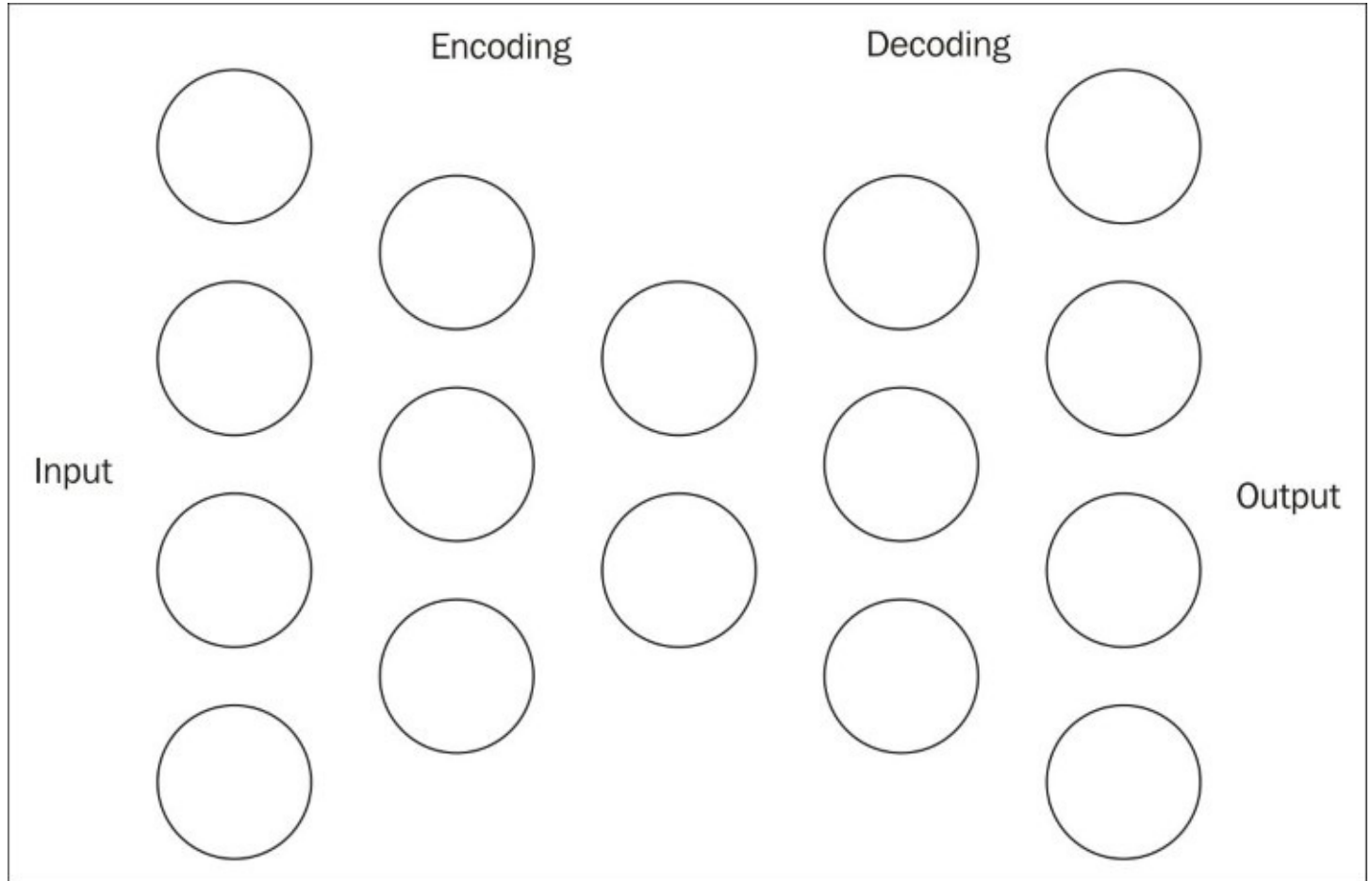
The purpose of the encoding sequence is to compress the original input into a smaller vector space. The middle layer of the previous figure is this compressed layer. These intermediate vectors can be thought of as possible features of the dataset. The encoding is also referred to as the pre-training half. It is the output of the intermediate RBM layer and does not perform classification.

The encoder's first layer will use more inputs than used by the dataset. This has the effect of expanding the features of the dataset. A sigmoid-belief unit is a form of non-linear transformation used with each layer. This unit is not able to accurately represent information as real values. However, using more inputs, it is able to do a better job.

The second half of the network performs decoding, effectively reconstructing the input. This is a

forward-feed network, using the same weights as the corresponding layers in the encoding half. However, the weights are transposed and are not initialized randomly. The training rate needs to be set lower for the second half.

An autoencoder is useful for data compression and searching. The output of the first half of the model is compressed, thus making it useful for storage and transmission usage. Later, it can be decompressed, as we will demonstrate in Chapter 10, *Visual and Audio Analysis*. This is sometimes referred to as semantic hashing.

If a series of inputs, such as images or sounds, have been compressed and stored, then new input can be compressed and matched with the stored values to find the best fit. An autoencoder can also be used for other information retrieval tasks.

Building an autoencoder in DL4J

This example is adapted from http://deeplearning4j.org/deepautoencoder. We start with a try-catch block to handle errors that may crop up and with a few variable declarations. This example uses the `Mnist` (http://yann.lecun.com/exdb/mnist/) dataset, which is a set of images containing hand-written numbers. Each image consists of `28` by `28` pixels. An iterator is declared to access the data:

```
try {
    final int numberOfRows = 28;
    final int numberOfColumns = 28;
    int seed = 123;
    int numberOfIterations = 1;

    iterator = new MnistDataSetIterator(
        1000, MnistDataFetcher.NUM_EXAMPLES, true);
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

Configuring the network

The configuration of the network is created using the `NeuralNetConfiguration.Builder()` class. Ten layers are created where the input layer consists of `1000` neurons. This is larger than the 28 by 28 pixel input and is used to compensate for the sigmoid-belief units used in each layer.

Each of the subsequent layers gets smaller until layer four is reached. This layer represents the last step of the encoding process. With layer five, the decoding process starts and the subsequent layers get bigger. The last layer uses `1000` neurons.

Each layer of the model uses an RBM instance except the last layer, which is constructed using the `OutputLayer.Builder` class. The configuration code follows:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(numberOfIterations)
        .optimizationAlgo(
          OptimizationAlgorithm.LINE_GRADIENT_DESCENT)
        .list()
        .layer(0, new RBM.Builder()
            .nIn(numberOfRows * numberOfColumns).nOut(1000)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(1, new RBM.Builder().nIn(1000).nOut(500)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(2, new RBM.Builder().nIn(500).nOut(250)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
            .build())
        .layer(3, new RBM.Builder().nIn(250).nOut(100)
            .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
```

```
                .build())
        .layer(4, new RBM.Builder().nIn(100).nOut(30)
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
                .build()) //encoding stops
        .layer(5, new RBM.Builder().nIn(30).nOut(100)
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
                .build()) //decoding starts
        .layer(6, new RBM.Builder().nIn(100).nOut(250)
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
                .build())
        .layer(7, new RBM.Builder().nIn(250).nOut(500)
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
                .build())
        .layer(8, new RBM.Builder().nIn(500).nOut(1000)
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT)
                .build())
        .layer(9, new OutputLayer.Builder(
                LossFunctions.LossFunction.RMSE_XENT).nIn(1000)
                .nOut(numberOfRows * numberOfColumns).build())
        .pretrain(true).backprop(true)
        .build();
```

Building and training the network

The model is then created and initialized, and score iteration listeners are set up:

```
model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(Collections.singletonList(
        (IterationListener) new ScoreIterationListener()));
```

The model is trained using the `fit` method:

```
while (iterator.hasNext()) {
    DataSet dataSet = iterator.next();
    model.fit(new DataSet(dataSet.getFeatureMatrix(),
            dataSet.getFeatureMatrix()));
}
```

Saving and retrieving a network

It is useful to save the model so that it can be used for later analysis. This is accomplished using the `ModelSerializer` class's `writeModel` method. It takes the `model` instance and `modelFile` instance, along with a `boolean` parameter specifying whether the model's updater should be saved. An updater is a learning algorithm used for adjusting certain model parameters:

```
modelFile = new File("savedModel");
ModelSerializer.writeModel(model, modelFile, true);
```

The model can be retrieved using the following code:

```
modelFile = new File("savedModel");
MultiLayerNetwork model =
ModelSerializer.restoreMultiLayerNetwork(modelFile);
```

Specialized autoencoders

There are specialized versions of autoencoders. When an autoencoder uses more hidden layers than inputs, it may learn the identity function, which is a function that always returns the same value used as input to the function. To avoid this problem, an extension to the **autoencoder**, **denoising autoencoder**, is used; it randomly modifies the input introducing noise. The amount of noise introduced varies depending on the input dataset. A **Stacked Denoising Autoencoder** (**SdA**) is a series of denoising autoencoders strung together.

Convolutional networks

CNNs are feed-forward networks modeled after the visual cortex found in animals. The visual cortex is arranged with overlapping neurons, and so in this type of network, the neurons are also arranged in overlapping sections, known as receptive fields. Due to their design model, they function with minimal preprocessing or prior knowledge, and this lack of human intervention makes them especially useful.

This type of network is used frequently in image and video recognition applications. They can be used for classification, clustering, and object recognition. CNNs can also be applied to text analysis by implementing **Optical Character Recognition** (**OCR**). CNNs have been a driving force in the machine learning movement in part due to their wide applicability in practical situations.

We are going to demonstrate a CNN using DL4J. The process will closely mirror the process we used in the *Building an autoencoder in DL4J* section. We will again use the `Mnist` dataset. This dataset contains image data, so it is well-suited to a convolutional network.

Building the model

First, we need to create a new `DataSetIterator` to process the data. The parameters for the `MnistDataSetIterator` constructor are the batch size, `1000` in this case, and the total number of samples to process. We then get our next dataset, shuffle the data to randomize, and split our data to be tested and trained. As we discussed earlier in the chapter, we typically use 65% of the data to train the data and the remaining 35% is used for testing:

```
DataSetIterator iter = new MnistDataSetIterator(1000,
MnistDataFetcher.NUM_EXAMPLES);
DataSet dataset = iter.next();
dataset.shuffle();
SplitTestAndTrain testAndTrain = dataset.splitTestAndTrain(0.65);
DataSet trainingData = testAndTrain.getTrain();
DataSet testData = testAndTrain.getTest();
```

We then normalize both sets of data:

```
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainingData);
normalizer.transform(trainingData);
normalizer.transform(testData);
```

Next, we can build our network. As shown earlier, we will again use a `MultiLayerConfiguration` instance with a series of `NeuralNetConfiguration.Builder` methods. We will discuss the individual methods after the following code sequence. Notice that the last layer again uses the `softmax` activation function for regression analysis:

```
MultiLayerConfiguration.Builder builder = new
        NeuralNetConfiguration.Builder()
    .seed(123)
    .iterations(1)
    .regularization(true).l2(0.0005)
    .weightInit(WeightInit.XAVIER)
    .optimizationAlgo(OptimizationAlgorithm
        .STOCHASTIC_GRADIENT_DESCENT)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list()
    .layer(0, new ConvolutionLayer.Builder(5, 5)
        .nIn(6)
        .stride(1, 1)
        .nOut(20)
        .activation("identity")
        .build())
    .layer(1, new SubsamplingLayer.Builder(
            SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .build())
    .layer(2, new ConvolutionLayer.Builder(5, 5)
        .stride(1, 1)
        .nOut(50)
        .activation("identity")
```

```
            .build())
    .layer(3, new SubsamplingLayer.Builder(
                SubsamplingLayer.PoolingType.MAX)
            .kernelSize(2, 2)
            .stride(2, 2)
            .build())
    .layer(4, new DenseLayer.Builder().activation("relu")
            .nOut(500).build())
    .layer(5, new OutputLayer.Builder(
                LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
            .nOut(10)
            .activation("softmax")
            .build())
    .backprop(true).pretrain(false);
```

The first layer, layer `0`, which is duplicated next for your convenience, uses the `ConvolutionLayer.Builder` method. The input to a convolution layer is the product of the image height, width, and number of channels. In a standard RGB image, there are three channels. The `nIn` method takes the number of channels. The `nOut` method specifies that `20` outputs are expected:

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        .nIn(6)
        .stride(1, 1)
        .nOut(20)
        .activation("identity")
        .build())
```

Layers `1` and `3` are both subsampling layers. These layers follow convolution layers and do no real convolution themselves. They return a single value, the maximum value for that input region:

```
.layer(1, new SubsamplingLayer.Builder(
        SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .build())
                    ...
.layer(3, new SubsamplingLayer.Builder(
        SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2, 2)
        .stride(2, 2)
        .build())
```

Layer `2` is also a convolution layer like layer `0`. Notice that we do not specify the number of channels in this layer:

```
.layer(2, new ConvolutionLayer.Builder(5, 5)
        .nOut(50)
        .activation("identity")
        .build())
```

The fourth layer uses the `DenseLayer.Builder` class, as in our earlier example. As mentioned previously, the `DenseLayer` class is a feed-forward and fully connected layer:

```
.layer(4, new DenseLayer.Builder().activation("relu")
        .nOut(500).build())
```

The layer 5 is an `OutputLayer` instance and uses `softmax` automation:

```
.layer(5, new OutputLayer.Builder(
            LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(10)
        .activation("softmax")
        .build())
        .backprop(true).pretrain(false);
```

Finally, we create a new instance of the `ConvolutionalLayerSetup` class. We pass the builder object and the dimensions of our image (28 x 28). We also pass the number of channels, in this case, `1`:

```
new ConvolutionLayerSetup(builder, 28, 28, 1);
```

We can now configure and fit our model. We once again use the `MultiLayerConfiguration` and `MultiLayerNetwork` classes to build our network. We set up listeners and then iterate through our data. For each `DataSet`, we execute the `fit` method:

```
MultiLayerConfiguration conf = builder.build();
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(Collections.singletonList((IterationListener)
  new ScoreIterationListener(1/5)));

while (iter.hasNext()) {
    DataSet next = iter.next();
    model.fit(new DataSet(next.getFeatureMatrix(), next.getLabels()));
}
```

We are now ready to evaluate our model.

Evaluating the model

To evaluate our model, we use the `Evaluation` class. We get the output from our model and send it, along with the labels for our dataset, to the `eval` method. We then execute the `stats` method to get the statistical information on our network:

```
Evaluation evaluation = new Evaluation(4);
INDArray output = model.output(testData.getFeatureMatrix());
evaluation.eval(testData.getLabels(), output);
out.println(evaluation.stats());
```

The following is a sample output from the execution of this code, for we are only showing the results of the `stats` method. The first part reports on how examples are classified and the second part displays various statistics:

```
Examples labeled as 0 classified by model as 0: 19 times
Examples labeled as 1 classified by model as 1: 41 times
Examples labeled as 2 classified by model as 1: 4 times
Examples labeled as 2 classified by model as 2: 30 times
Examples labeled as 2 classified by model as 3: 1 times
Examples labeled as 3 classified by model as 2: 1 times
Examples labeled as 3 classified by model as 3: 28 times
==========================Scores================================Accuracy:
0.3371
Precision: 0.8481
Recall: 0.8475
F1 Score: 0.8478
================================================================
```

As in our previous model, the evaluation demonstrates decent accuracy and success with our network.

Recurrent Neural Networks

RNN differ from feed-forward networks in that their input includes the input from the previous iteration or step. They still process the current input but use a feedback loop to take into consideration the inputs to the prior step, also called the recent past, for context. This step effectively gives the network memory. One popular type of recurrent network involves **Long Short-Term Memory** (**LSTM**). This type of memory improves the processing power of the network.

RNNs are designed to process sequential data and are especially useful for analysis and prediction with text data. Given a sequence of words, an RNN can predict the probability of each word being the next in the sequence. This also allows for text generation by the network. RNNs are versatile and also process image data well, especially image labeling applications. The flexibility in design and purpose and ease in training make RNNs popular choices for many data science applications. DL4J also provides support for LSTM networks and other RNNs.

Summary

In this chapter, we examined deep learning techniques for neural networks. All API support in this chapter was provided by Deeplearning4j. We began by demonstrating how to acquire and prepare data for use with deep learning networks. We discussed how to configure and build a model. This was followed by an explanation of how to train and test a model by splitting the dataset into training and testing segments.

Our discussion continued with an examination of deep learning and regression analysis. We showed how to prepare the data and class, build the model, and evaluate the model. We used sample data and displayed output statistics to demonstrate the relative effectiveness of our model.

RBM and DBNs were then examined. DBNs are comprised of RBMs stacked together and are especially useful for classification and clustering applications. Deep autoencoders are also built using RBMs, with two symmetrical DBNs. The autoencoders are especially useful for feature selection and extraction.

Finally, we demonstrated a convolutional network. This network is modeled after the visual cortex and allows the network to use regions of information for classification. As in previous examples, we built, trained, and then evaluated the model for effectiveness. We then concluded the chapter with a brief introduction to recurrent neural networks.

We will expand upon these topics as we move into next chapter and examine text analysis techniques.

Chapter 9. Text Analysis

Text analysis is a broad topic and is typically referred to as **Natural Language Processing** (**NLP**). It is used for many different tasks, including text searching, language translation, sentiment analysis, speech recognition, and classification, to mention a few. The process of analyzing can be difficult due to the particularities and ambiguity found in natural languages. However, there has been a considerable amount of work in this area and there are several Java APIs supporting this effort.

We will start with an introduction to the basic concepts and tasks used in NLP. These include the following:

- **Tokenization**: The process of splitting text into individual tokens or words.
- **Stop words**: These are words that are common and may not be necessary for processing. They include such words as the, a, and to.
- **Name Entity Recognition** (**NER**): This is the process of identifying elements of text such as people's name, locations, or things.
- **Parts of Speech** (**POS**): This identifies the grammatical parts of a sentence such as noun, verb, adjective, and so on.
- **Relationships**: Here, we are concerned with identifying how parts of text are related to each other, such as the subject and object of a sentence.

The concepts of words, sentences, and paragraphs are well known. However, extracting and analyzing these components is not always that straightforward. The term **corpus** frequently refers to a collection of text.

As with most data science problems, it is important to preprocess text. Frequently, this involves handling such tasks as these:

- Handling Unicode
- Converting text to uppercase or lowercase
- Removing stop words

We examined several techniques for tokenization and removing stop words in Chapter 3, *Data Cleaning*. In this chapter, we will focus on POS, NER, extracting relationships from sentence, text classification, and sentiment analysis.

There are several NLP APIs available, including these:

- **OpenNLP** (https://opennlp.apache.org/): An open source Apache project
- **StanfordNLP** (http://nlp.stanford.edu/software/) : Another open source library
- **UIMA** (https://uima.apache.org/): An Apache project supporting pipelines
- **LingPipe** (http://alias-i.com/lingpipe/): A library that uses pipelines extensively
- **DL4J** (http://deeplearning4j.org/): The Deep Learning for Java library supports various classes for deep learning neural networks including support for NLP

We will use OpenNLP and DL4J to demonstrate text analysis in this chapter. We chose these because they are both well-known and have good published resources for additional support.

We will use the Google **Word2Vec** and **Doc2Vec** neural networks to perform text classification. This includes feature vectors based on other words as well as using labeled information to classify documents. Finally, we will discuss sentiment analysis. This type of analysis seeks to assign meaning to text and also uses the Word2Vec network.

We start our discussion with NER.

Implementing named entity recognition

This is sometimes referred to as finding people and things. Given a text segment, we may want to identify all the names of people present. However, this is not always easy because a name such as Rob may also be used as a verb.

In this section, we will demonstrate how to use OpenNLP's `TokenNameFinderModel` class to find names and locations in text. While there are other entities we may want to find, this example will demonstrate the basics of the technique. We begin with names.

Most names occur within a single line. We do not want to use multiple lines because an entity such as a state might inadvertently be identified incorrectly. Consider the following sentences:

Jim headed north. Dakota headed south.

If we ignored the period, then the state of North Dakota might be identified as a location, when in fact it is not present.

Using OpenNLP to perform NER

We start our example with a try-catch block to handle exceptions. OpenNLP uses models that have been trained on different sets of data. In this example, the `en-token.bin` and `en-ner-person.bin` files contain the models for the tokenization of English text and for English name elements, respectively. These files can be downloaded from http://opennlp.sourceforge.net/models-1.5/. However, the IO stream used here is standard Java:

```
try (InputStream tokenStream =
            new FileInputStream(new File("en-token.bin"));
        InputStream personModelStream = new FileInputStream(
            new File("en-ner-person.bin"));) {
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

An instance of the `TokenizerModel` class is initialized using the token stream. This instance is then used to create the actual `TokenizerME` tokenizer. We will use this instance to tokenize our sentence:

```
TokenizerModel tm = new TokenizerModel(tokenStream);
TokenizerME tokenizer = new TokenizerME(tm);
```

The `TokenNameFinderModel` class is used to hold a model for name entities. It is initialized using the person model stream. An instance of the `NameFinderME` class is created using this model since we are looking for names:

```
TokenNameFinderModel tnfm = new
  TokenNameFinderModel(personModelStream);
NameFinderME nf = new NameFinderME(tnfm);
```

To demonstrate the process, we will use the following sentence. We then convert it to a series of tokens using the tokenizer and `tokenizer` method:

```
String sentence = "Mrs. Wilson went to Mary's house for dinner.";
String[] tokens = tokenizer.tokenize(sentence);
```

The `Span` class holds information regarding the positions of entities. The `find` method will return the position information, as shown here:

```
Span[] spans = nf.find(tokens);
```

This array holds information about person entities found in the sentence. We then display this information as shown here:

```
for (int i = 0; i < spans.length; i++) {
    out.println(spans[i] + " - " + tokens[spans[i].getStart()]);
}
```

The output for this sequence is as follows. Notice that it identifies the last name of Mrs. Wilson but not the "Mrs.":

```
[1..2) person - Wilson
[4..5) person - Mary
```

Once these entities have been extracted, we can use them for specialized analysis.

Identifying location entities

We can also find other types of entities such as dates and locations. In the following example, we find locations in a sentence. It is very similar to the previous person example, except that an `en-ner-location.bin` file is used for the model:

```
try (InputStream tokenStream =
            new FileInputStream("en-token.bin");
        InputStream locationModelStream = new FileInputStream(
            new File("en-ner-location.bin"));) {

    TokenizerModel tm = new TokenizerModel(tokenStream);
    TokenizerME tokenizer = new TokenizerME(tm);

    TokenNameFinderModel tnfm =
        new TokenNameFinderModel(locationModelStream);
    NameFinderME nf = new NameFinderME(tnfm);

    sentence = "Enid is located north of Oklahoma City.";
    String tokens[] = tokenizer.tokenize(sentence);

    Span spans[] = nf.find(tokens);

    for (int i = 0; i < spans.length; i++) {
        out.println(spans[i] + " - " +
        tokens[spans[i].getStart()]);
    }
} catch (Exception ex) {
    // Handle exceptions
}
```

With the sentence defined previously, the model was only able to find the second city, as shown here. This likely due to the confusion that arises with the name `Enid` which is both the name of a city and a person' name:

[5..7) location - Oklahoma

Suppose we use the following sentence:

```
sentence = "Pond Creek is located north of Oklahoma City.";
```

Then we get this output:

[1..2) location - Creek
[6..8) location - Oklahoma

Unfortunately, it has missed the town of `Pond Creek`. NER is a useful tool for many applications, but like many techniques, it is not always foolproof. The accuracy of the NER approach presented, and many of the other NLP examples, will vary depending on factors such as the accuracy of the model, the language being used, and the type of entity.

We may also be interested in how text can be classified. We will examine one approach in the next section.

Classifying text

Classifying text is an important part of machine learning and data science. We have to be able to classify text for a variety of applications, including document retrieval and web searches. It is often important to assign specific labels to the data before we can determine its usefulness for a particular application or search result.

In this chapter, we are going to demonstrate a technique involving the use of paragraph vectors and labeled data with DL4J classes. This example allows us to read in documents and, based on the text inside of the document, assign a label (or classification) to the document. We are also going to show an example of classifying text by similarity. This means we will match phrases and words that have similar structure. This example will also use DL4J.

Word2Vec and Doc2Vec

We will be using Word2Vec and Doc2Vec in a few examples in this chapter. Word2Vec is a neural network with two layers used for text processing. Given a body of text, the network will provide feature vectors for the words contained in the text. These vectors are simply mathematical representations of the word features and can be numerically compared to other vectors. This comparison is often referred to as the distance between two words.

Word2Vec operates with the understanding that words can be classified by determining the probability that two words will occur together. Because of this methodology, Word2Vec can be used for more than classification of sentences. Any object or data that can be represented by text labels can be classified with this network.

Doc2Vec is an extension of Word2Vec. Rather than building vectors representing the features of individual words compared to other words, as Word2Vec does, this network compares words to given labels. The vectors are set up to represent the theme or overall meaning of a document. Our next example shows how these feature vectors are then associated with specific documents.

Classifying text by labels

In our first example using Doc2Vec, we will associate our documents with three labels: health, finance, and science. But before we can associate the data with labels, we have to define those labels and train our model to recognize the labels. Each label represents the meaning or classification of a particular piece of text.

In this example we will use sample documents, each pre-labelled with our categories: health, finance, or science. We will use these paragraphs to train our model and then, as in previous examples, use a set of test data to test our model. We will be using the files found at https://github.com/deeplearning4j/dl4j-examples/tree/master/dl4j-examples/src/main/resources/paravec. We have based this example upon sample code written for DL4J, which can be found at https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/nlp/paragraphvectors/ParagraphVectorsClas

First we need to set up some instance variables to use later in our code. We will be using a `ParagraphVectors` object to create our vectors, a `LabelAwareIterator` object to iterate through our data, and a `TokenizerFactory` object to tokenize our data:

```
ParagraphVectors pVect;
LabelAwareIterator iter;
TokenizerFactory tFact;
```

Then we will set up our `ClassPathResource`. This specifies the directory within our project that contains the data files to be classified. The first resource contains our labeled data used for training purposes. We then direct our iterator and tokenizer to use the resources specified as the `ClassPathResource`. We also specify that we will use the `CommonPreprocessor` to preprocess our data:

```
ClassPathResource resource = new
        ClassPathResource("paravec/labeled");

iter = new FileLabelAwareIterator.Builder()
        .addSourceFolder(resource.getFile())
        .build();

tFact = new DefaultTokenizerFactory();
tFact.setTokenPreProcessor(new CommonPreprocessor());
```

Next, we build our `ParagraphVectors`. This is where we specify the learning rate, batch size, and number of training epochs. We include our iterator and tokenizer in the setup process as well. Once we've built our `ParagraphVectors`, we call the `fit` method to train our model using the training data in the `paravec/labeled` directory:

```
pVect = new ParagraphVectors.Builder()
        .learningRate(0.025)
        .minLearningRate(0.001)
        .batchSize(1000)
```

```
            .epochs(20)
            .iterate(iter)
            .trainWordVectors(true)
            .tokenizerFactory(tFact)
            .build();

pVect.fit();
```

Now that we have trained our model, we can use our unlabeled data to test. We create a new `ClassPathResource` for our unlabeled data and create a new `FileLabelAwareIterator` as well:

```
ClassPathResource unlabeledText =
        new ClassPathResource("paravec/unlabeled");
FileLabelAwareIterator unlabeledIter =
        new FileLabelAwareIterator.Builder()
            .addSourceFolder(unlabeledText.getFile())
            .build();
```

The next step involves iterating through our unlabeled data and identifying the correct label for each document. We can generally expect that each document will fall into multiple labels but have a different weight, or percent match, for each. So, while one article may be mostly classified as a health article, it likely has enough information to be also classified, to a lesser degree, as a science article.

Next, we set up a `MeansBuilder` and `LabelSeeker` object. These classes access tables containing the relationships between words and labels, which we will use in our `ParagraphVectors`. The `InMemoryLookupTable` class provides access to a default table for word lookup:

```
MeansBuilder mBuilder =
    new MeansBuilder((InMemoryLookupTable<VocabWord>)
      pVect.getLookupTable(),tFact);
LabelSeeker lSeeker =
    new LabelSeeker(iter.getLabelsSource().getLabels(),
              (InMemoryLookupTable<VocabWord>)
    pVect.getLookupTable());
```

Finally, we iterate through our unlabeled documents. For each document, we will change the document into a vector and use our `LabelSeeker` to get the scores for each document. We log the scores for each document and print out the score with the appropriate labels:

```
while (unlabeledIter.hasNextDocument()) {
    LabelledDocument doc = unlabeledIter.nextDocument();
    INDArray docCentroid = mBuilder.documentAsVector(doc);
    List<Pair<String, Double>> scores =
            lSeeker.getScores(docCentroid);
    out.println("Document '" + doc.getLabel() +
      "' falls into the following categories: ");
    for (Pair<String, Double> score : scores) {
      out.println ("        " + score.getFirst() + ": " +
          score.getSecond());
```

```
        }

}
```

The output from our preceding print statements is as follows:

```
Document 'finance' falls into the following categories:
finance: 0.2889593541622162
health: 0.11753179132938385
science: 0.021202782168984413
Document 'health' falls into the following categories:
finance: 0.059537000954151154
health: 0.27373185753822327
science: 0.07699354737997055
```

In each instance, our documents were classified properly, as demonstrated by the higher percentage assigned to the correct label category. This classification can be used in conjunction with other data analysis techniques to draw additional conclusions about the data contained in the files. Often text classification is an initial or early step in a data analysis process as documents are classified into groups for further analysis.

Classifying text by similarity

In this next example, we will match different text samples based on their structure and similarity. We will still be using the `ParagraphVectors` class we used in the previous example. To begin, download the `raw_sentences.txt` file from GitHub (https://github.com/deeplearning4j/dl4j-examples/tree/master/dl4j-examples/src/main/resources) and add it to your project. This file contains a list of sentences which we will read in, label, and then compare.

First, we set up our `ClassPathResource` and assign an iterator to handle our file data. We have used a `SentenceIterator` for this example:

```
ClassPathResource srcFile = new
      ClassPathResource("/raw_sentences.txt");
File file = srcFile.getFile();
SentenceIterator iter = new BasicLineIterator(file);
```

Next, we will again use `TokenizerFactory` to tokenize our data. We also want to create a new `LabelsSource` object. This allows us to define the format of our sentence labels. We have chosen to prefix each line with `LINE_`:

```
TokenizerFactory tFact = new DefaultTokenizerFactory();
tFact.setTokenPreProcessor(new CommonPreprocessor());
LabelsSource labelFormat = new LabelsSource("LINE_");
```

Now we are ready to build our `ParagraphVectors`. Our setup process includes these methods: `minWordFrequency`, which specifies the minimum word frequency to use in the training corpus, and `iterations`, which specifies the number of iterations for each mini batch. We also set the number of epochs, the layer size, and the learning rate. Additionally, we include our `LabelsSource`, defined before, and our iterator and tokenizer. The `trainWordVectors` method specifies whether word and document representations should be built together. Finally, `sampling` determines whether subsampling should occur or not. We then call our `build` and `fit` methods:

```
ParagraphVectors vec = new ParagraphVectors.Builder()
        .minWordFrequency(1)
        .iterations(5)
        .epochs(1)
        .layerSize(100)
        .learningRate(0.025)
        .labelsSource(labelFormat)
        .windowSize(5)
        .iterate(iter)
        .trainWordVectors(false)
        .tokenizerFactory(tFact)
        .sampling(0)
        .build();

vec.fit();
```

Next, we will include some statements to evaluate the accuracy of our classifications. It is important to note that while the document itself starts at 1, the indexing process begins at 0. So, for example, line 9836 in the document will be associated with the label LINE_9835. We will first compare three sentences that should be classified as somewhat similar, and then two examples comparing dissimilar sentences. The similarity method takes two labels and returns the relative distance between them in the form of double:

```java
double similar1 = vec.similarity("LINE_9835", "LINE_12492");
out.println("Comparing lines 9836 & 12493
        ('This is my house .'/'This is my world .')
        Similarity = " + similar1);


double similar2 = vec.similarity("LINE_3720", "LINE_16392");
out.println("Comparing lines 3721 & 16393
        ('This is my way .'/'This is my work .')
        Similarity = " + similar2);

double similar3 = vec.similarity("LINE_6347", "LINE_3720");
out.println("Comparing lines 6348 & 3721
        ('This is my case .'/'This is my way .')
        Similarity = " + similar3);

double dissimilar1 = vec.similarity("LINE_3720", "LINE_9852");
out.println("Comparing lines 3721 & 9853
        ('This is my way .'/'We now have one .')
        Similarity = " + dissimilar1);

double dissimilar2 = vec.similarity("LINE_3720", "LINE_3719");
out.println("Comparing lines 3721 & 3720
        ('This is my way .'/'At first he says no .')
        Similarity = " + dissimilar2);
```

The output of our print statements is shown as follows. Compare the result of the similarity method for the three similar sentences and the two dissimilar sentences. Of particular note, the similarity method result for the last example, two very dissimilar sentences, returned a negative number. This implies a more significant disparity:

```
16:56:15.423 [main] INFO o.d.m.s.SequenceVectors - Epoch: [1]; Words
vectorized so far: [3171540]; Lines vectorized so far: [485810];
learningRate: [1.0E-4]
Comparing lines 9836 & 12493 ('This is my house .'/'This is my world .')
Similarity = 0.7641470432281494
Comparing lines 3721 & 16393 ('This is my way .'/'This is my work .')
Similarity = 0.7246013879776001
Comparing lines 6348 & 3721 ('This is my case .'/'This is my way .')
Similarity = 0.8988922834396362
Comparing lines 3721 & 9853 ('This is my way .'/'We now have one .')
Similarity = 0.5840312242507935
Comparing lines 3721 & 3720 ('This is my way .'/'At first he says no .')
Similarity = -0.6491150259971619
```

Although this example uses ParagraphVectors like our first classification example, this

demonstrates flexibility in our approach. We can use these DL4J libraries to classify data in more than one manner.

Understanding tagging and POS

POS is concerned with identifying the types of components found in a sentence. For example, this sentence has several elements, including the verb "has", several nouns such as "example" and "elements", and adjectives such as "several". Tagging, or more specifically **POS tagging**, is the process of associating element types to words.

POS tagging is useful as it adds more information about the sentence. We can ascertain the relationship between words and often their relative importance. The results of tagging are often used in later processing steps.

This task can be difficult as we are unable to rely upon a simple dictionary of words to determine their type. For example, the word `lead` can be used as both a noun and as a verb. We might use it in either of the following two sentences:

```
He took the lead in the play.
Lead the way!
```

POS tagging will attempt to associate the proper label to each word of a sentence.

Using OpenNLP to identify POS

To illustrate this process, we will be using OpenNLP (https://opennlp.apache.org/). This is an open source Apache project which supports many other NLP processing tasks.

We will be using the `POSModel` class, which can be trained to recognize POS elements. In this example, we will use it with a previously trained model based on the **Penn TreeBank tag-set** (http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html). Various pretrained models are found at http://opennlp.sourceforge.net/models-1.5/. We will be using the `en-pos-maxent.bin` model. This has been trained on English text using what is called maximum entropy.

Maximum entropy refers to the amount of uncertainty in the model which it maximizes. For a given problem there is a set of probabilities describing what is known about the data set. These probabilities are used to build a model. For example, we may know that there is a 23 percent chance that one specific event may follow a certain condition. We do not want to make any assumptions about unknown probabilities so we avoid adding unjustified information. A maximum entropy approach attempts to preserve as much uncertainty as possible; hence it attempts to maximize entropy.

We will also use the `POSTaggerME` class, which is a maximum entropy tagger. This is the class that will make tag predictions. With any sentence, there may be more than one way of classifying, or tagging, its components.

We start with code to acquire the previously trained English tagger model and a simple sentence to be tagged:

```
try (InputStream input = new FileInputStream(
        new File("en-pos-maxent.bin"));) {
    String sentence = "Let's parse this sentence.";
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The tagger uses an array of strings, where each string is a word. The following sequence takes the previous sentence and creates an array called `words`. The first part uses the `Scanner` class to parse the sentence string. We could have used other code to read the data from a file if needed. After that, the `List` class's `toArray` method is used to create the array of strings:

```
List<String> list = new ArrayList<>();
Scanner scanner = new Scanner(sentence);
while(scanner.hasNext()) {
    list.add(scanner.next());
}
String[] words = new String[1];
words = list.toArray(words);
```

The model is then built using the file containing the model:

```
POSModel posModel = new POSModel(input);
```

The tagger is then created based on the model:

```
POSTaggerME posTagger = new POSTaggerME(posModel);
```

The `tag` method does the actual work. It is passed an array of words and returns an array of tags. The words and tags are then displayed:

```
String[] posTags = posTagger.tag(words);
for(int i=0; i<posTags.length; i++) {
    out.println(words[i] + " - " + posTags[i]);
}
```

The output for this example follows:

```
Let's - NNP
parse - NN
this - DT
sentence. - NN
```

The analysis has determined that the word `let's` is a singular proper noun while the words `parse` and `sentence` are singular nouns. The word `this` is a determiner, that is, it is a word that modifies another and helps identify a phrase as general or specific. A list of tags is provided in the next section.

Understanding POS tags

The POS elements returned abbreviations. A list of **Penn TreeBankPOS** tags can be found at
https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html. The following
is a shortened version of this list:

Tag	Description	Tag	Description
DT	Determiner	RB	Adverb
JJ	Adjective	RBR	Adverb, comparative
JJR	Adjective, comparative	RBS	Adverb, superlative
JJS	Adjective, superlative	RP	Particle
NN	Noun, singular or mass	SYM	Symbol
NNS	Noun, plural	TOP	Top of the parse tree
NNP	Proper noun, singular	VB	Verb, base form
NNPS	Proper noun, plural	VBD	Verb, past tense
POS	Possessive ending	VBG	Verb, gerund or present participle
PRP	Personal pronoun	VBN	Verb, past participle
PRP$	Possessive pronoun	VBP	Verb, non-3rd person singular present
S	Simple declarative clause	VBZ	Verb, 3rd person singular present

As mentioned earlier, there may be more than one possible set of POS assignments for a sentence.
The `topKSequences` method, as shown next, will return various assignment possibilities along
with a score. The method returns an array of `Sequence` objects whose `toString` method returns
the score and POS list:

```
Sequence sequences[] = posTagger.topKSequences(words);
for(Sequence sequence : sequences) {
    out.println(sequence);
}
```

The output for the previous sentence follows, where the last sequence is considered to be the most probable alternative:

```
-2.3264880694837213 [NNP, NN, DT, NN]
-2.6610271245387853 [NNP, VBD, DT, NN]
-2.6630142638557217 [NNP, VB, DT, NN]
```

Each line of output assigns possible tags to each word of the sentence. We can see that only the second word, parse, is determined to have other possible tags.

Next, we will demonstrate how to extract relationships from text.

Extracting relationships from sentences

Knowing the relationship between elements of a sentence is important in many analysis tasks. It is useful for assessing the important content of a sentence and providing insight into the meaning of a sentence. This type of analysis has been used for tasks ranging from grammar checking to speech recognition to language translations.

In the previous section, we demonstrated one approach used to extract the parts of speech. Using this technique, we were able to identify the sentence element types present in a sentence. However, the relationships between these elements is missing. We need to parse the sentence to extract these relationships between sentence elements.

Using OpenNLP to extract relationships

There are several techniques and APIs that can be used to extract this type of information. In this section we will use OpenNLP to demonstrate one way of extracting the structure of a sentence. The demonstration is centered around the `ParserTool` class, which uses a previously trained model. The parsing process will return the probabilities that the sentence's elements extracted are correct. As will many NLP tasks, there are often multiple answers possible.

We start with a try-with-resource block to open an input stream for the model. The `en-parser-chunking.bin` file contains a model that uses parses text into its POS. In this case, it is trained for English:

```
try (InputStream modelInputStream = new FileInputStream(
            new File("en-parser-chunking.bin"));) {
    ...
} catch (Exception ex) {
    // Handle exceptions
}
```

Within the try block an instance of the `ParserModel` class is created using the input stream. The actual parser is created next using the `ParserFactory` class's `create` method:

```
ParserModel parserModel = new ParserModel(modelInputStream);
Parser parser = ParserFactory.create(parserModel);
```

We will use the following sentence to test the parser. The `ParserTool` class's `parseLine` method does the actual parsing and returns an array of `Parse` objects. Each of these objects holds one parsing alternative. The last argument of the `parseLine` method specifies how many alternatives to return:

```
String sentence = "Let's parse this sentence.";
Parse[] parseTrees = ParserTool.parseLine(sentence, parser, 3);
```

The next sequence displays each of the possibilities:

```
for(Parse tree : parseTrees) {
    tree.show();
}
```

The output of the show method for this example follows. The tags were previously defined in *Understanding POS tags* section:

```
(TOP (NP (NP (NNP Let's) (NN parse)) (NP (DT this) (NN sentence.))))
(TOP (S (NP (NNP Let's)) (VP (VB parse) (NP (DT this) (NN sentence.)))))
(TOP (S (NP (NNP Let's)) (VP (VBD parse) (NP (DT this) (NN sentence.)))))
```

The following example reformats the last two outputs to better show the relationships. They differ in how they classify the verb parse:

```
(TOP
```

```
(S
(NP (NNP Let's))
(VP (VB parse)
(NP (DT this) (NN sentence.))
)
)
)
(TOP
(S
(NP (NNP Let's))
(VP (VBD parse)
(NP (DT this) (NN sentence.))
)
)
)
```

When there are multiple parse alternatives, the `Parse` class's `getProb` returns a probability that reflects the model's confidence in the alternatives. The following sequence demonstrates this method:

```
for(Parse tree : parseTrees) {
    out.println("Probability: " + tree.getProb());
}
```

The output follows:

```
Probability: -3.6810244423259078
Probability: -3.742475884515823
Probability: -4.16148634555491
```

Another interesting NLP task is sentiment analysis, which we will demonstrate next.

Sentiment analysis

Sentiment analysis involves the evaluation and classification of words based on their context, meaning, and emotional implications. Typically, if we were to look up a word in a dictionary we will find a meaning or definition for the word but, taken out of the context of a sentence, we may not be able to ascribe detailed and precise meaning to the word.

For example, the word toast could be defined as simply a slice of heated and browned bread. But in the context of the sentence *He's toast!*, the meaning changes completely. Sentiment analysis seeks to derive meanings of words based on their context and usage.

It is important to note that advanced sentiment analysis will expand beyond simple positive or negative classification and ascribe detailed emotional meaning to words. It is far simpler to classify words as positive or negative but far more useful to classify them as happy, furious, indifferent, or anxious.

This type of analysis falls into the category of effective computing, a type of computing interested in the emotional implications and uses of technological tools. This type of computing is especially significant given the growing amount of emotionally influenced data readily available for analysis on social media sites today.

Being able to determine the emotional content of text enables a more targeted, and appropriate response. For example, being able to judge the emotional response in a chat session between a customer and technical representative can allow the representative to do a better job. This can be especially important when there is a cultural or language gap between them.

This type of analysis can also be applied to visual images. It could be used to gauge someone's response to a new product, such as when conducting a taste test, or to judge how people react to scenes of s movie or commercial.

As part of our example we will be using a bag-of-words model. Bag-of-words models simplify word representation for natural language processing by containing a set, known as the **bag**, of words irrespective of grammar or word order. The words have features used for classification, most importantly the frequency of each word. Because some words such as the, a, or and will naturally have a higher frequency in any text, the words are given a weight as well. Common words with less contextual significance will have a smaller weight and factor less into the text analysis.

Downloading and extracting the Word2Vec model

To demonstrate sentiment analysis, we will use Google's Word2Vec models in conjunction with DL4J to simply classify movie reviews as either positive or negative based upon the words used in the review. This example is adapted from work done by Alex Black (https://github.com/deeplearning4j/dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/recurrent/word2vecsentiment/Word2VecSen). As discussed previously in this chapter, Word2Vec consists of two-layer neural networks trained to build meaning from the context of words. We will also be using a large set of movie reviews from http://ai.stanford.edu/~amaas/data/sentiment/.

Before we begin, you will need to download the Word2Vec data from https://code.google.com/p/word2vec/. The basic process includes:

- Downloading and extracting the movie reviews
- Loading the Word2Vec Google News vectors
- Loading each movie review

The words within the reviews are then broken into vectors and used to train the network. We will train the network across five epochs and evaluate the network's performance after each epoch.

To begin, we first declare three final variables. The first is the URL to retrieve the training data, the second is the location to store our extracted data, and the third is the location of the Google News vectors on the local machine. Modify this third variable to reflect the location on your local machine:

```
public static final String TRAINING_DATA_URL =
    "http://ai.stanford.edu/~amaas/" +
    "data/sentiment/aclImdb_v1.tar.gz";
public static final String EXTRACT_DATA_PATH =
    FilenameUtils.concat(System.getProperty(
    "java.io.tmpdir"), "dl4j_w2vSentiment/");
public static final String GNEWS_VECTORS_PATH =
    "C:/YOUR_PATH/GoogleNews-vectors-negative300.bin" +
    "/GoogleNews-vectors-negative300.bin";
```

Next we download and extract our model data. The next two methods are modelled after the code found in the DL4J example. We first create a new method, `getModelData`. The method is shown next in its entirety.

First we create a new `File` using the `EXTRACT_DATA_PATH` we defined previously. If the file does not already exist, we create a new directory. Next, we create two more `File` objects, one for the path to the archived TAR file and one for the path to the extracted data. Before we attempt to extract the data, we check whether these two files exist. If the archive path does not exist, we download the data from the `TRAINING_DATA_URL` and then extract the data. If the extracted file does not exist, we then extract the data:

```
private static void getModelData() throws Exception {
    File modelDir = new File(EXTRACT_DATA_PATH);
    if (!modelDir.exists()) {
        modelDir.mkdir();
    }
    String archivePath = EXTRACT_DATA_PATH + "aclImdb_v1.tar.gz";
    File archiveName = new File(archivePath);
    String extractPath = EXTRACT_DATA_PATH + "aclImdb";
    File extractName = new File(extractPath);
    if (!archiveName.exists()) {
        FileUtils.copyURLToFile(new URL(TRAINING_DATA_URL),
                archiveName);
        extractTar(archivePath, EXTRACT_DATA_PATH);
    } else if (!extractName.exists()) {
        extractTar(archivePath, EXTRACT_DATA_PATH);
    }
}
```

To extract our data, we will create another method called `extractTar`. We will provide two inputs to the method, the `archivePath` and the `EXTRACT_DATA_PATH` defined before. We also need to define our buffer size to use in the extraction process:

```
private static final int BUFFER_SIZE = 4096;
```

We first create a new `TarArchiveInputStream`. We use the `GzipCompressorInputStream` because it provides support for extracting `.gz` files. We also use the `BufferedInputStream` to improve performance in our extraction process. The compressed file is very large and may take some time to download and extract.

Next we create a `TarArchiveEntry` and begin reading in data using the `TarArchiveInputStream` `getNextEntry` method. As we process entry in the compressed file, we first check whether the entry is a directory. If it is, we create a new directory in our extraction location. Finally we create a new `FileOutputStream` and `BufferedOutputStream` and use the `write` method to write our data in the extracted location:

```
private static void extractTar(String dataIn, String dataOut)
    throws IOException {
        try (TarArchiveInputStream inStream =
            new TarArchiveInputStream(
                new GzipCompressorInputStream(
                    new BufferedInputStream(
                        new FileInputStream(dataIn))))) {
            TarArchiveEntry tarFile;
            while ((tarFile = (TarArchiveEntry) inStream.getNextEntry())
                != null) {
                if (tarFile.isDirectory()) {
                    new File(dataOut + tarFile.getName()).mkdirs();
                }else {
                    int count;
                    byte data[] = new byte[BUFFER_SIZE];
                    FileOutputStream fileInStream =
                      new FileOutputStream(dataOut + tarFile.getName());
```

```
                    BufferedOutputStream outStream =
                      new BufferedOutputStream(fileInStream,
                        BUFFER_SIZE);
                    while ((count = inStream.read(data, 0, BUFFER_SIZE))
                        != -1) {
                            outStream.write(data, 0, count);
                    }
                }
            }
        }
    }
```

Building our model and classifying text

Now that we have created methods to download and extract our data, we need to declare and initialize variables used to control the execution of our model. Our `batchSize` refers to the amount of words we process in each example, in this case `50`. Our `vectorSize` determines the size of the vectors. The Google News model has word vectors of size `300`. `nEpochs` refers to the number of times we attempt to run through our training data. Finally, `truncateReviewsToLength` specifies whether, for memory utilization purposes, we should truncate the movie reviews if they exceed a specific length. We have chosen to truncate reviews longer than `300` words:

```
int batchSize = 50;
int vectorSize = 300;
int nEpochs = 5;
int truncateReviewsToLength = 300;
```

Now we can set up our neural network. We will use a `MultiLayerConfiguration` network, as discussed in Chapter 8, *Deep Learning* . In fact, our example here is very similar to the model built in configuring and building a model, with a few differences. In particular, in this model we will use a faster learning rate and a `GravesLSTM` recurrent network in layer 0. We will have the same number of input neurons as we have words in our vector, in this case, `300`. We also use `gradientNormalization`, a technique used to help our algorithm find the optimal solution. Notice we are using the `softmax` activation function, which was discussed in Chapter 8, *Deep Learning* . This function uses regression and is especially suited for classification algorithms:

```
MultiLayerConfiguration sentimentNN =
        new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm
            .STOCHASTIC_GRADIENT_DESCENT).iterations(1)
    .updater(Updater.RMSPROP)
    .regularization(true).l2(1e-5)
    .weightInit(WeightInit.XAVIER)
    .gradientNormalization(GradientNormalization
            .ClipElementWiseAbsoluteValue)
            .gradientNormalizationThreshold(1.0)
    .learningRate(0.0018)
    .list()
    .layer(0, new GravesLSTM.Builder()
            .nIn(vectorSize).nOut(200)
          .activation("softsign").build())
    .layer(1, new RnnOutputLayer.Builder()
            .activation("softmax")
            .lossFunction(LossFunctions.LossFunction.MCXENT)
            .nIn(200).nOut(2).build())
    .pretrain(false).backprop(true).build();
```

We can then create our `MultiLayerNetwork`, initialize the network, and set listeners.

```
MultiLayerNetwork net = new MultiLayerNetwork(sentimentNN);
net.init();
net.setListeners(new ScoreIterationListener(1));
```

Next we create a `WordVectors` object to load our Google data. We use a `DataSetIterator` to test and train our data. The `AsyncDataSetIterator` allows us to load our data in a separate thread, to improve performance. This process requires a large amount of memory and so improvements such as this are essential for optimal performance:

```
WordVectors wordVectors = WordVectorSerializer
DataSetIterator trainData = new AsyncDataSetIterator(
    new SentimentExampleIterator(EXTRACT_DATA_PATH, wordVectors,
        batchSize, truncateReviewsToLength, true), 1);
DataSetIterator testData = new AsyncDataSetIterator(
    new SentimentExampleIterator(EXTRACT_DATA_PATH, wordVectors,
        100, truncateReviewsToLength, false), 1);
```

Finally, we are ready to train and evaluate our data. We run through our data `nEpochs` times; in this case, we have five iterations. Each iteration executes the `fit` method against our training data and then creates a new `Evaluation` object to evaluate our model using `testData`. The evaluation is based on around 25,000 movie reviews and can take a significant amount to time to run. As we evaluate the data, we create `INDArray` to store information, including the feature matrix and labels from our data. This data is used later in the `evalTimeSeries` method for evaluation. Finally, we print out our evaluation statistics:

```
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);
    trainData.reset();

    Evaluation evaluation = new Evaluation();
    while (testData.hasNext()) {
        DataSet t = testData.next();
        INDArray dataFeatures = t.getFeatureMatrix();
        INDArray dataLabels = t.getLabels();
        INDArray inMask = t.getFeaturesMaskArray();
        INDArray outMask = t.getLabelsMaskArray();
        INDArray predicted = net.output(dataFeatures, false,
            inMask, outMask);

        evaluation.evalTimeSeries(dataLabels, predicted, outMask);
    }
    testData.reset();

    out.println(evaluation.stats());
}
```

The output from the final iteration is shown next. Our examples classified as `0` are considered negative reviews and the ones classified as `1` are considered positive reviews:

```
Epoch 4 complete. Starting evaluation:
Examples labeled as 0 classified by model as 0: 11122 times
Examples labeled as 0 classified by model as 1: 1378 times
Examples labeled as 1 classified by model as 0: 3193 times
Examples labeled as 1 classified by model as 1: 9307 times
==========================Scores================================Accuracy:
0.8172
Precision: 0.824
```

```
Recall: 0.8172
F1 Score: 0.8206
======================================================================
```

If compared with previous iterations, you should notice the score and accuracy improving with each evaluation. With each iteration, our model improves its accuracy in classifying movie reviews as either negative or positive.

Summary

In this chapter, we introduced a number of NLP tasks and showed how they are supported. In particular, we used OpenNLP and DL4J to illustrate how they are performed. While there are a number of other libraries available, these examples provide a good introduction to the techniques.

We started with an introduction to basic NLP terms and concepts such as named entity recognition, POS, and relationships between elements of a sentence. Named entity recognition is concerned with finding and labeling the parts of a sentence such as people, locations, and things. POS associates labels with elements of a sentence. For example, NN refers to a noun and VB to a verb.

We then included a discussion of the Word2Vec and Doc2Vec neural networks. These were used to classify text, both with labels and by similarity with other words. We demonstrated the use of DL4J resources to create feature vectors for document association with labels.

While the identification of these associations is interesting, a more useful analysis is performed when relationships are extracted from a sentence. We demonstrated how relationships are found using OpenNLP. The POS are associated with each word and the relationships between the words are shown using a set of tags and parentheses. This type of analysis can be used for more sophisticated analyses such as language translation and grammar checking.

Finally, we discussed and showed examples of sentiment analysis. This process involves classifying text based on its tone or contextual meaning. We examined a process for classifying movie reviews as positive or negative.

In this chapter, we demonstrated various techniques for text analysis and classification. In our next chapter, we will examine techniques designed for video and audio analysis.

Chapter 10. Visual and Audio Analysis

The use of sound, images, and videos is becoming a more important aspect of our day-to-day lives. Phone conversations and devices reliant on voice commands are increasingly common. People regularly conduct video chats with other people around the world. There has been a rapid proliferation of photo and video sharing sites. Applications that utilize images, video, and sound from a variety of sources are becoming more common.

In this chapter, we will demonstrate several techniques available to Java to process sounds and images. The first part of the chapter addresses sound processing. Both speech recognition and **Text-To-Speech** (**TTS**) APIs will be demonstrated. Specifically, we will use the FreeTTS (http://freetts.sourceforge.net/docs/index.php) API to convert text to speech, followed with a demonstration of the CMU Sphinx toolkit for speech recognition.

The **Java Speech API** (**JSAPI**) (http://www.oracle.com/technetwork/java/index-140170.html) provides access to speech technology. It is not part of the standard JDK but is supported by third-party vendors. Its intent is to support speech recognition and speech synthesizers. There are several vendors that support JSAPI, including FreeTTS and Festival (http://www.cstr.ed.ac.uk/projects/festival/).

In addition, there are several cloud-based speech APIs, including IBM's support through **Watson Cloud** speech-to-text capabilities.

Next, we will examine image processing techniques, including facial recognition. This involves identifying faces within an image. This technique is easy to accomplish using OpenCV (http://opencv.org/) which we will demonstrate in the Identifying faces section.

We will end the chapter with a discussion of Neuroph Studio, a neural network Java-based editor, to classify images and perform image recognition. We will continue to use faces here and attempt to train a network to recognize images of human faces.

Text-to-speech

Speech synthesis generates human speech. TTS converts text to speech and is useful for a number of different applications. It is used in many places, including phone help desk systems and ordering systems. The TTS process typically consists of two parts. The first part tokenizes and otherwise processes the text into speech units. The second part converts these units into speech.

The two primary approaches for TTS uses **concatenation synthesis** and **formant synthesis**. Concatenation synthesis frequently combines prerecorded human speech to create the desired output. Formant synthesis does not use human speech but generates speech by creating electronic waveforms.

We will be using FreeTTS (http://freetts.sourceforge.net/docs/index.php) to demonstrate TTS. The latest version can be downloaded from https://sourceforge.net/projects/freetts/files/. This approach uses concatenation to generate speech.

There are several important terms used in TTS/FreeTTS:

- **Utterance** - This concept corresponds roughly to the vocal sounds that make up a word or phrase
- **Items** - Sets of features (name/value pairs) that represent parts of an utterance
- **Relationship** - A list of items, used by FreeTTS to iterate back and forward through an utterance
- **Phone** - A distinct sound
- **Diphone** - A pair of adjacent phones

The FreeTTS Programmer's Guide (http://freetts.sourceforge.net/docs/ProgrammerGuide.html) details the process of converting text to speech. This is a multi-step process whose major steps include the following:

- **Tokenization** - Extracting the tokens from the text
- **TokenToWords** - Converting certain words, such as 1910 to nineteen ten
- **PartOfSpeechTagger** - This step currently does nothing, but is intended to identify the parts of speech
- **Phraser** - Creates a phrase relationship for the utterance
- **Segmenter** - Determines where syllable breaks occur
- **PauseGenerator** - This step inserts pauses within speech, such as before utterances
- **Intonator** - Determines the accents and tones
- **PostLexicalAnalyzer** - This step fixes problems such as a mismatch between the available diphones and the one that needs to be spoken
- **Durator** - Determines the duration of the syllables
- **ContourGenerator** - Calculates the fundamental frequency curve for an utterance, which maps the frequency against time and contributes to the generation of tones
- **UnitSelector** - Groups related diphones into a unit
- **PitchMarkGenerator** - Determines the pitches for an utterance

- **UnitConcatenator** - Concatenates the diphone data together

The following figure is from the *FreeTTS Programmer's Guide, Figure 11:* The **Utterance** after **UnitConcatenator** processing, and depicts the process. This high-level overview of the TTS process provides a hint at the complexity of the process:
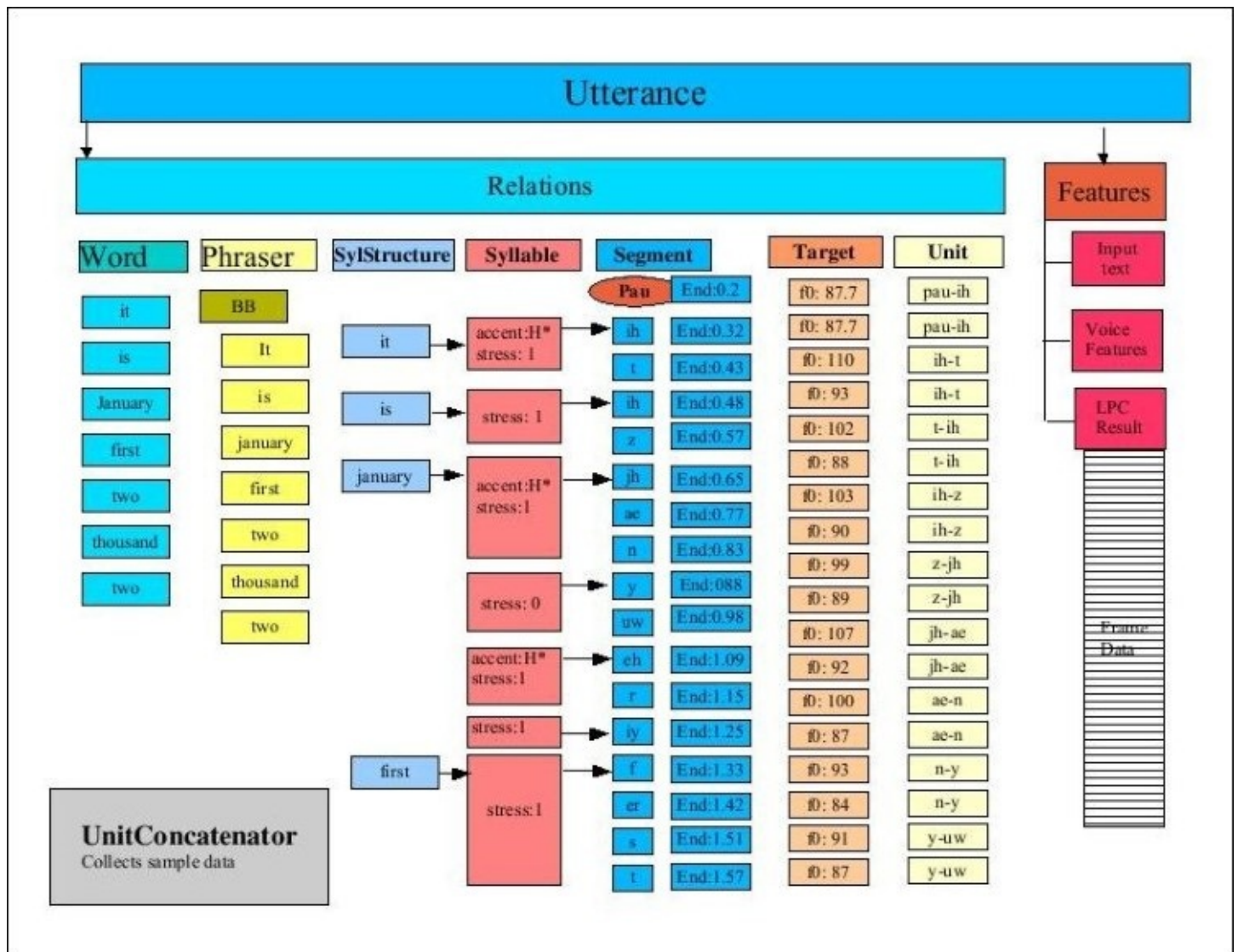
Using FreeTTS

TTS system facilitates the use of different voices. For example, these differences may be in the language, the sex of the speaker, or the age of the speaker.

The **MBROLA Project's** (http://tcts.fpms.ac.be/synthesis/mbrola.html) objective is to support voice synthesizers for as many languages as possible. MBROLA is a speech synthesizer that can be used with a TTS system such as FreeTTS to support TTS synthesis.

Download MBROLA for the appropriate platform binary from http://tcts.fpms.ac.be/synthesis/mbrola.html. From the same page, download any desired MBROLA voices found at the bottom of the page. For our examples we will use usa1, usa2, and usa3. Further details about the setup are found at http://freetts.sourceforge.net/mbrola/README.html.

The following statement illustrates the code needed to access the MBROLA voices. The setProperty method assigns the path where the MBROLA resources are found:

```
System.setProperty("mbrola.base", "path-to-mbrola-directory");
```

To demonstrate how to use TTS, we use the following statement. We obtain an instance of the VoiceManager class, which will provide access to various voices:

```
VoiceManager voiceManager = VoiceManager.getInstance();
```

To use a specific voice the getVoice method is passed the name of the voice and returns an instance of the Voice class. In this example, we used mbrola_us1, which is a US English, young, female voice:

```
Voice voice = voiceManager.getVoice("mbrola_us1");
```

Once we have obtained the Voice instance, use the allocate method to load the voice. The speak method is then used to synthesize the words passed to the method as a string, as illustrated here:

```
voice.allocate();
voice.speak("Hello World");
```

When executed, the words "Hello World" should be heard. Try this with other voices, as described in the next section, and text to see which combination is best suited for an application.

Getting information about voices

The `VoiceManager` class' `getVoices` method is used to obtain an array of the voices currently available. This can be useful to provide users with a list of voices to choose from. We will use the method here to illustrate some of the voices available. In the next code sequence, the method returns the array, whose elements are then displayed:

```
Voice[] voices = voiceManager.getVoices();
for (Voice v : voices) {
    out.println(v);
}
```

The output will be similar to the following:

CMUClusterUnitVoice
CMUDiphoneVoice
CMUDiphoneVoice
MbrolaVoice
MbrolaVoice
MbrolaVoice

The `getVoiceInfo` method provides potentially more useful information, though it is somewhat verbose:

```
out.println(voiceManager.getVoiceInfo());
```

The first part of the output follows; the `VoiceDirectory` directory is displayed followed by the details of the voice. Notice that the directory name contains the name of the voice. The `KevinVoiceDirectory` contains two voices: `kevin` and `kevin16`:

VoiceDirectory 'com.sun.speech.freetts.en.us.cmu_time_awb.AlanVoiceDirectory'
Name: alan
Description: default time-domain cluster unit voice
Organization: cmu
Domain: time
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 12.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
VoiceDirectory 'com.sun.speech.freetts.en.us.cmu_us_kal.KevinVoiceDirectory'
Name: kevin
Description: default 8-bit diphone voice
Organization: cmu
Domain: general
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT

```
Pitch: 100.0
Pitch Range: 11.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
Name: kevin16
Description: default 16-bit diphone voice
Organization: cmu
Domain: general
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 11.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
...
```

Using voices from a JAR file

Voices can be stored in JAR files. The `VoiceDirectory` class provides access to voices stored in this manner. The voice directories available to FreeTTs are found in the **lib** directory and include the following:

- `cmu_time_awb.jar`
- `cmu_us_kal.jar`

The name of a voice directory can be obtained from the command prompt:

```
java -jar fileName.jar
```

For example, execute the following command:

```
java -jar cmu_time_awb.jar
```

It generates the following output:

```
VoiceDirectory 'com.sun.speech.freetts.en.us.cmu_time_awb.AlanVoiceDirectory'
Name: alan
Description: default time-domain cluster unit voice
Organization: cmu
Domain: time
Locale: en_US
Style: standard
Gender: MALE
Age: YOUNGER_ADULT
Pitch: 100.0
Pitch Range: 12.0
Pitch Shift: 1.0
Rate: 150.0
Volume: 1.0
```

Gathering voice information

The `Voice` class provides a number of methods that permit the extraction or setting of speech characteristics. As we demonstrated earlier, the `VoiceManager` class' `getVoiceInfo` method provided information about the voices currently available. However, we can use the `Voice` class to get information about a specific voice.

In the following example, we will display information about the voice `kevin16`. We start by getting an instance of this `voice` using the `getVoice` method:

```
VoiceManager vm = VoiceManager.getInstance();
Voice voice = vm.getVoice("kevin16");
voice.allocate();
```

Next, we call a number of the `Voice` class' `get` method to obtain specific information about the voice. This includes previous information provided by the `getVoiceInfo` method and other information that is not otherwise available:

```
out.println("Name: " + voice.getName());
out.println("Description: " + voice.getDescription());
out.println("Organization: " + voice.getOrganization());
out.println("Age: " + voice.getAge());
out.println("Gender: " + voice.getGender());
out.println("Rate: " + voice.getRate());
out.println("Pitch: " + voice.getPitch());
out.println("Style: " + voice.getStyle());
```

The output of this example follows:

```
Name: kevin16
Description: default 16-bit diphone voice
Organization: cmu
Age: YOUNGER_ADULT
Gender: MALE
Rate: 150.0
Pitch: 100.0
Style: standard
```

These results are self-explanatory and give you an idea of the type of information available. There are additional methods that give you access to details regarding the TTS process that are not normally of interest. This includes information such as the audio player being used, utterance-specific data, and features of a specific phone.

Having demonstrated how text can be converted to speech, we will now examine how we can convert speech to text.

Understanding speech recognition

Converting speech to text is an important application feature. This ability is increasingly being used in a wide variety of contexts. Voice input is used to control smart phones, automatically handle input as part of help desk applications, and to assist people with disabilities, to mention a few examples.

Speech consists of an audio stream that is complex. Sounds can be split into **phones**, which are sound sequences that are similar. Pairs of these phones are called **diphones**. **Utterances** consist of words and various types of pauses between them.

The essence of the conversion process involves splitting sounds by silences between utterances. These utterances are then matched to the words that most closely sound like the utterance. However, this can be difficult due to many factors. For example, these differences may be in the form of variances in how words are pronounced due to the context of the word, regional dialects, the quality of the sound, and other factors.

The matching process is quite involved and often uses multiple models. A model may be used to match acoustic features with a sound. A phonetic model can be used to match phones to words. Another model is used to restrict word searches to a given language. These models are never entirely accurate and contribute to inaccuracies found in the recognition process.

We will be using CMUSphinx 4 to illustrate this process.

Using CMUPhinx to convert speech to text

Audio processed by CMUSphinx must be in **Pulse Code Modulation** (**PCM**) format. PCM is a technique that samples analog data, such as an analog wave representing speech, and produces a digital version of the signal. FFmpeg (https://ffmpeg.org/) is a free tool that can convert between audio formats if needed.

You will need to create sample audio files using the PCM format. These files should be fairly short and can contain numbers or words. It is recommended that you run the examples with different files to see how well the speech recognition works.

First, we set up the basic framework for the conversion by creating a try-catch block to handle exceptions. First, create an instance of the `Configuration` class. It is used to configure the recognizer to recognize standard English. The configuration models and dictionary need to be changed to handle other languages:

```
try {
    Configuration configuration = new Configuration();
    String prefix = "resource:/edu/cmu/sphinx/models/en-us/";
    configuration
            .setAcousticModelPath(prefix + "en-us");
    configuration
            .setDictionaryPath(prefix + "cmudict-en-us.dict");
    configuration
            .setLanguageModelPath(prefix + "en-us.lm.bin");
    ...
} catch (IOException ex) {
    // Handle exceptions
}
```

The `StreamSpeechRecognizer` class is then created using `configuration`. This class processes the speech based on an input stream. In the following code, we create an instance of the `StreamSpeechRecognizer` class and an `InputStream` from the speech file:

```
StreamSpeechRecognizer recognizer = new StreamSpeechRecognizer(
        configuration);
InputStream stream = new FileInputStream(new File("filename"));
```

To start speech processing, the `startRecognition` method is invoked. The `getResult` method returns a `SpeechResult` instance that holds the result of the processing. We then use the `SpeechResult` method to get the best results. We stop the processing using the `stopRecognition` method:

```
recognizer.startRecognition(stream);
SpeechResult result;
while ((result = recognizer.getResult()) != null) {
    out.println("Hypothesis: " + result.getHypothesis());
}
recognizer.stopRecognition();
```

When this is executed, we get the following, assuming the speech file contained this sentence:

Hypothesis: mary had a little lamb

When speech is interpreted there may be more than one possible word sequence. We can obtain the best ones using the `getNbest` method, whose argument specifies how many possibilities should be returned. The following demonstrates this method:

```
Collection<String> results = result.getNbest(3);
for (String sentence : results) {
    out.println(sentence);
}
```

One possible output follows:

<s> mary had a little lamb </s>
<s> marry had a little lamb </s>
<s> mary had a a little lamb </s>

This gives us the basic results. However, we will probably want to do something with the actual words. The technique for getting the words is explained next.

Obtaining more detail about the words

The individual words of the results can be extracted using the `getWords` method, as shown next. The method returns a list of `WordResult` instance, each of which represents one word:

```
List<WordResult> words = result.getWords();
for (WordResult wordResult : words) {
    out.print(wordResult.getWord() + " ");
}
```

The output for this code sequence follows `<sil>` reflects a silence found at the beginning of the speech:

`<sil> mary had a little lamb`

We can extract more information about the words using various methods of the `WordResult` class. In this sequence that follows, we will return the confidence and time frame associated with each word.

The `getConfidence` method returns the confidence expressed as a log. We use the `SpeechResult` class' `getResult` method to get an instance of the `Result` class. Its `getLogMath` method is then used to get a `LogMath` instance. The `logToLinear` method is passed the confidence value and the value returned is a real number between 0 and 1.0 inclusive. More confidence is reflected by a larger value.

The `getTimeFrame` method returns a `TimeFrame` instance. Its `toString` method returns two integer values, separated by a colon, reflecting the beginning and end times of the word:

```
for (WordResult wordResult : words) {
    out.printf("%s\n\tConfidence: %.3f\n\tTime Frame: %s\n",
            wordResult.getWord(), result
                    .getResult()
                    .getLogMath()
                    .logToLinear((float)wordResult
                            .getConfidence()),
            wordResult.getTimeFrame());
}
```

One possible output follows:

```
<sil>
Confidence: 0.998
Time Frame: 0:430
mary
Confidence: 0.998
Time Frame: 440:900
had
Confidence: 0.998
Time Frame: 910:1200
a
Confidence: 0.998
```
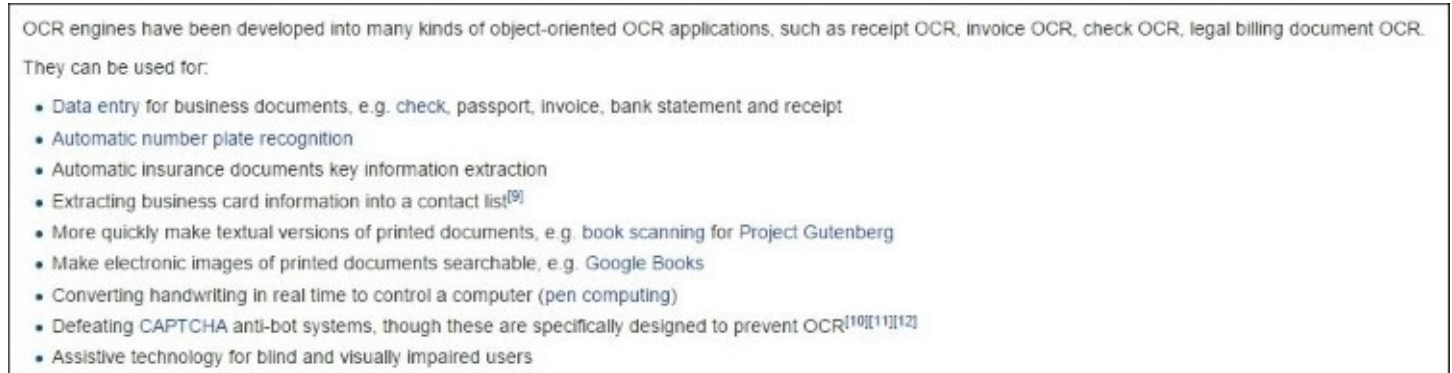
```
Time Frame: 1210:1340
little
Confidence: 0.998
Time Frame: 1350:1680
lamb
Confidence: 0.997
Time Frame: 1690:2170
```

Now that we have examined how sound can be processed, we will turn our attention to image processing.

Extracting text from an image

The process of extracting text from an image is called **O ptical Character Recognition** (**OCR**). This can be very useful when the text data that needs to be processed is embedded in an image. For example, the information contained in license plates, road signs, and directions can be very useful at times.

We can perform OCR using Tess4j (http://tess4j.sourceforge.net/), a Java JNA wrapper for Tesseract OCR API. We will demonstrate how to use the API using an image captured from the Wikipedia article on OCR (https://en.wikipedia.org/wiki/Optical_character_recognition#Applications). The Javadoc for the API is found at http://tess4j.sourceforge.net/docs/docs-3.0/. The image we use is shown here:

OCR engines have been developed into many kinds of object-oriented OCR applications, such as receipt OCR, invoice OCR, check OCR, legal billing document OCR.

They can be used for:

- Data entry for business documents, e.g. check, passport, invoice, bank statement and receipt
- Automatic number plate recognition
- Automatic insurance documents key information extraction
- Extracting business card information into a contact list[9]
- More quickly make textual versions of printed documents, e.g. book scanning for Project Gutenberg
- Make electronic images of printed documents searchable, e.g. Google Books
- Converting handwriting in real time to control a computer (pen computing)
- Defeating CAPTCHA anti-bot systems, though these are specifically designed to prevent OCR[10][11][12]
- Assistive technology for blind and visually impaired users

Using Tess4j to extract text

The `ITesseract` interface contains numerous OCR methods. The `doOCR` method takes a file and returns a string containing the words found in the file, as shown here:

```
ITesseract instance = new Tesseract();
try {
    String result = instance.doOCR(new File("OCRExample.png"));
    out.println(result);
} catch (TesseractException e) {
    // Handle exceptions
}
```

Part of the output is shown next:

```
OCR engines nave been developed into many lunds oiobiectorlented OCR
applicatlons, sucn as reoeipt OCR, involoe OCR, check OCR, legal billing
document OCR
They can be used ior
- Data entry ior business documents, e g check, passport, involoe, bank
statement and receipt
- Automatic number plate recognnlon
```

As you can see, there are numerous errors in this example. Often the quality of an image needs to be improved before it can be processed correctly. Techniques for improving the quality of the output can be found at https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality. For example, we can use the `setLanguage` method to specify the language processed. Also, the method often works better on TIFF images.

In the next example, we used an enlarged portion of the previous image, as shown here:

The output is much better, as shown here:

```
OCR engines have been developed into many kinds of object-oriented OCR
applications, such as receipt OCR,
invoice OCR, check OCR, legal billing document OCR.
They can be used for:
. Data entry for business documents, e.g. check, passport, invoice, bank
statement and receipt
. Automatic number plate recognition
```

These examples highlight the need for the careful cleaning of data.

Identifying faces

Identifying faces within an image is useful in many situations. It can potentially classify an image as one containing people or find people in an image for further processing. We will use OpenCV 3.1 (http://opencv.org/opencv-3-1.html) for our examples.

OpenCV (http://opencv.org/) is an open source computer vision library that supports several programming languages, including Java. It supports a number of techniques, including machine learning algorithms, to perform computer vision tasks. The library supports such operations as face detection, tracking camera movements, extracting 3D models, and removing red eye from images. In this section, we will demonstrate face detection.

Using OpenCV to detect faces

The example that follows was adapted from
http://docs.opencv.org/trunk/d9/d52/tutorial_java_dev_intro.html. Start by loading the native
libraries added to your system when OpenCV was installed. On Windows, this requires that
appropriate DLL files are available:

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

We used a base string to specify the location of needed OpenCV files. Using an absolute path
works better with many methods:

```
String base = "PathToResources";
```

The `CascadeClassifier` class is used for object classification. In this case, we will use it for
face detection. An XML file is used to initialize the class. In the following code, we use the
`lbpcascade_frontalface.xml` file, which provides information to assist in the identification of
objects. In the OpenCV download are several files, as listed here, that can be used for specific
face recognition scenarios:

- `lbpcascade_frontalcatface.xml`
- `lbpcascade_frontalface.xml`
- `lbpcascade_frontalprofileface.xml`
- `lbpcascade_silverware.xml`

The following statement initializes the class to detect faces:

```
CascadeClassifier faceDetector =
        new CascadeClassifier(base +
            "/lbpcascade_frontalface.xml");
```

The image to be processed is loaded, as shown here:

```
Mat image = Imgcodecs.imread(base + "/images.jpg");
```

For this example, we used the following image:

To find this image, perform a Google search using the term **people**. Select the **Images** category and then filter for **Labeled for reuse**. The image has the label: **Closeup portrait of a group of business people laughing by Lynda Sanchez**.

When faces are detected, the location within the image is stored in a `MatOfRect` instance. This class is intended to hold vectors and matrixes for any faces found:

```
MatOfRect faceVectors = new MatOfRect();
```

At this point, we are ready to detect faces. The `detectMultiScale` method performs this task. The image and the `MatOfRect` instance to hold the locations of any images are passed to the method:

```
faceDetector.detectMultiScale(image, faceVectors);
```

The next statement shows how many faces were detected:

```
out.println(faceVectors.toArray().length + " faces found");
```

We need to use this information to augment the image. This process will draw boxes around each face found, as shown next. To do this, the `Imgproc` class' `rectangle` method is used. The method is called once for each face detected. It is passed the image to be modified and the points represented the boundaries of the face:

```
for (Rect rect : faceVectors.toArray()) {
    Imgproc.rectangle(image, new Point(rect.x, rect.y),
            new Point(rect.x + rect.width, rect.y + rect.height),
            new Scalar(0, 255, 0));
}
```

The last step writes this image to a file using the `Imgcodecs` class' `imwrite` method:

```
Imgcodecs.imwrite("faceDetection.png", image);
```

As shown in the following image, it was able to identify four images:
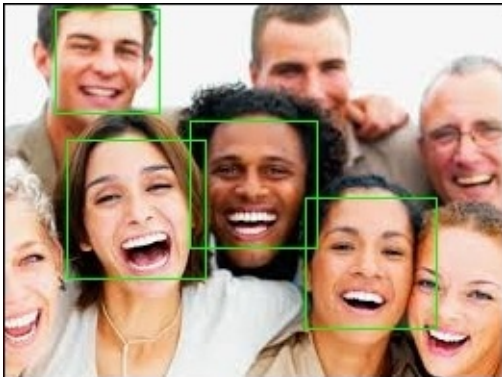
Using different configuration files will work better for other facial profiles.

Classifying visual data

In this section, we will demonstrate one technique for classifying visual data. We will use Neuroph to accomplish this. Neuroph is a Java-based neural network framework that supports a variety of neural network architectures. Its open source library provides support and plugins for other applications. In this example, we will use its neural network editor, Neuroph Studio, to create a network. This network can be saved and used in other applications. Neuroph Studio is available for download here: http://neuroph.sourceforge.net/download.html. We are building upon the process shown here: http://neuroph.sourceforge.net/image_recognition.htm.

For our example, we will create a **Multi Layer Perceptron** (**MLP**) network. We will then train our network to recognize images. We can both train and test our network using Neuroph Studio. It is important to understand how MLP networks recognize and interpret image data. Every image is basically represented by three two-dimensional arrays. Each array contains information about the color components: one array contains information about the color red, one about the color green, and one about the color blue. Every element of the array holds information about one specific pixel in the image. These arrays are then flattened into a one-dimensional array to be used as an input by the neural network.

# Creating a Neuroph Studio project for classifying visual images

To begin, first create a new Neuroph Studio project:
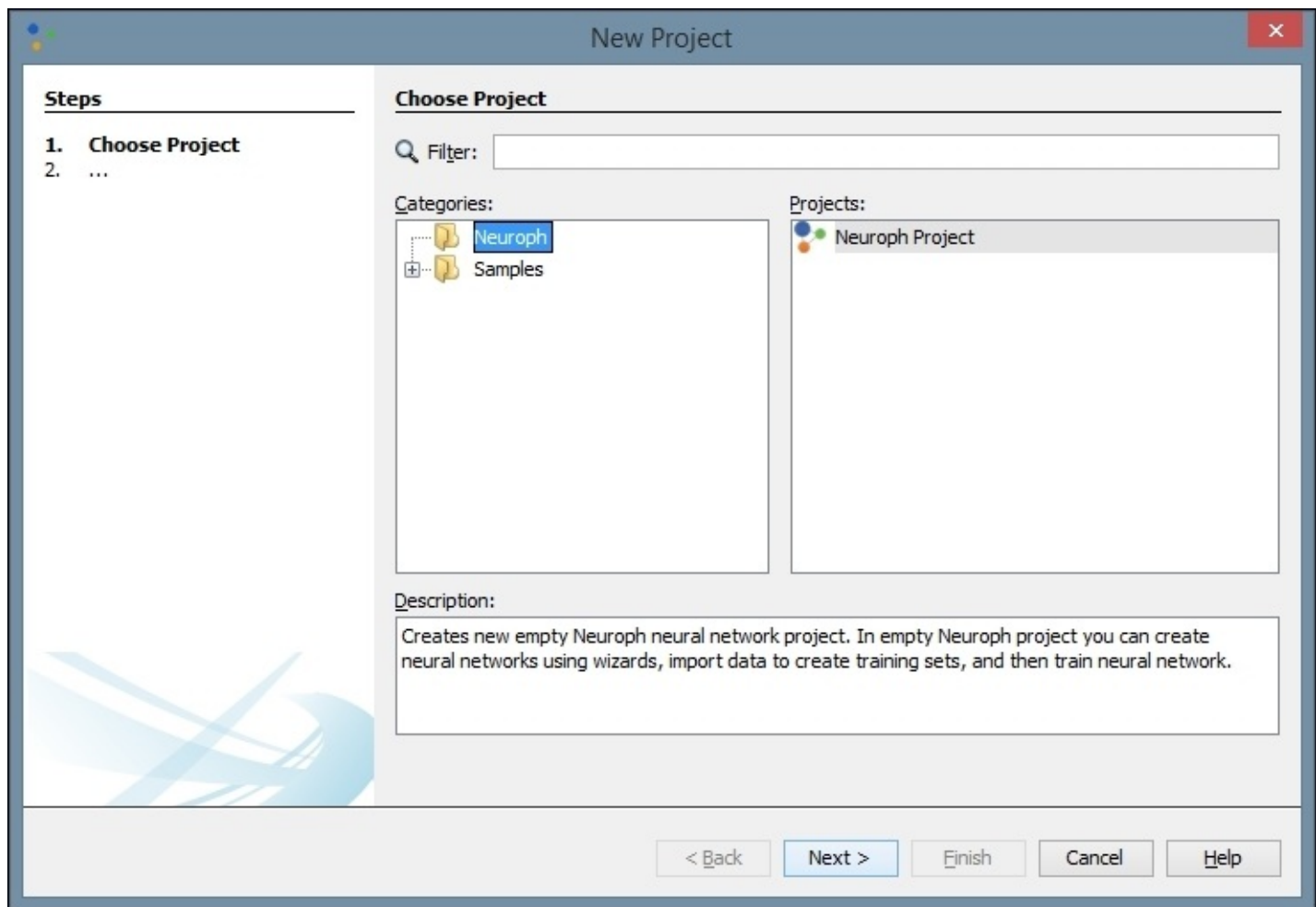
We will name our project `RecognizeFaces` because we are going train the neural network to recognize images of human faces:
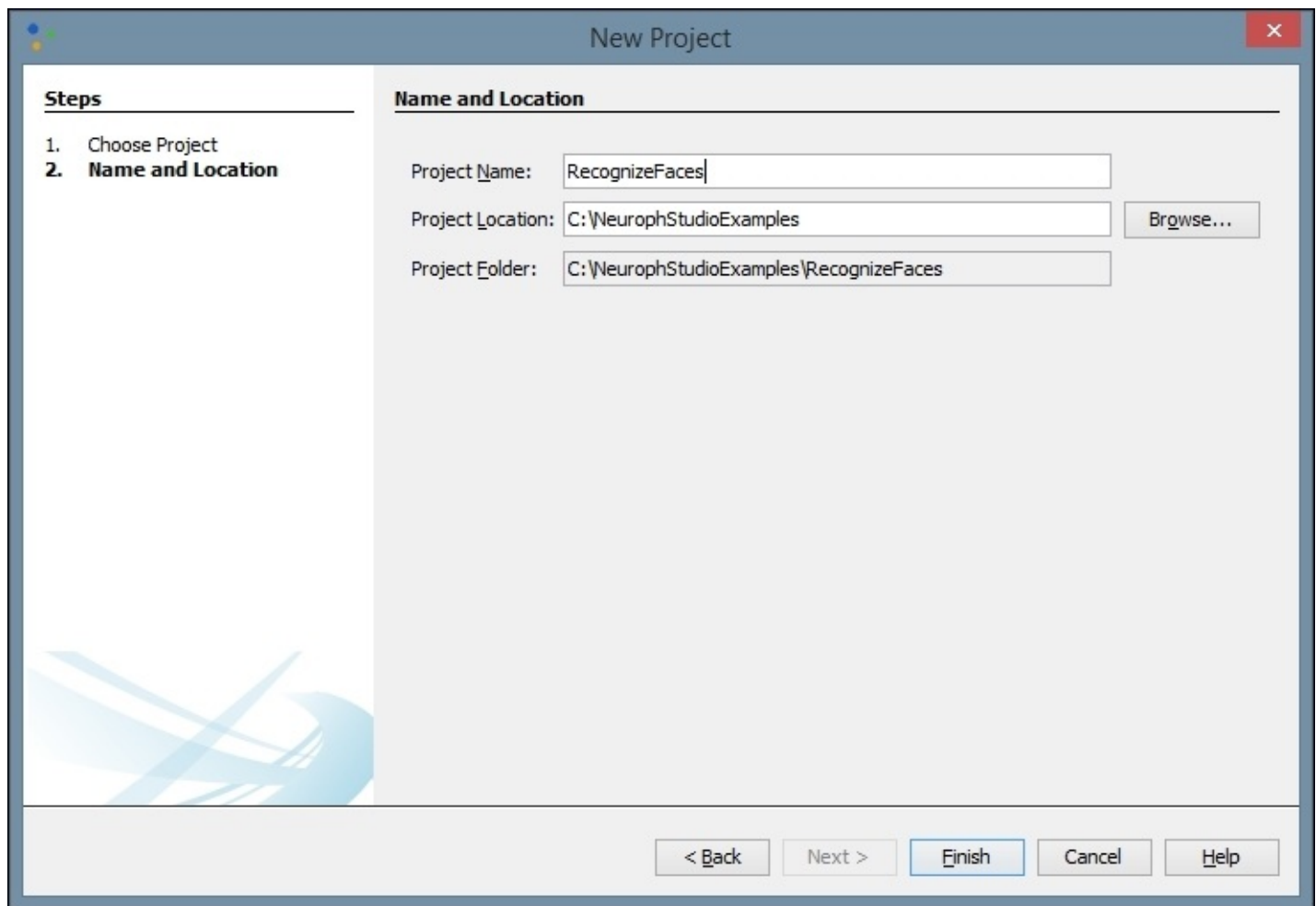
Next, we create a new file in our project. There are many types of project to choose from, but we will choose an **Image Recognition** type:
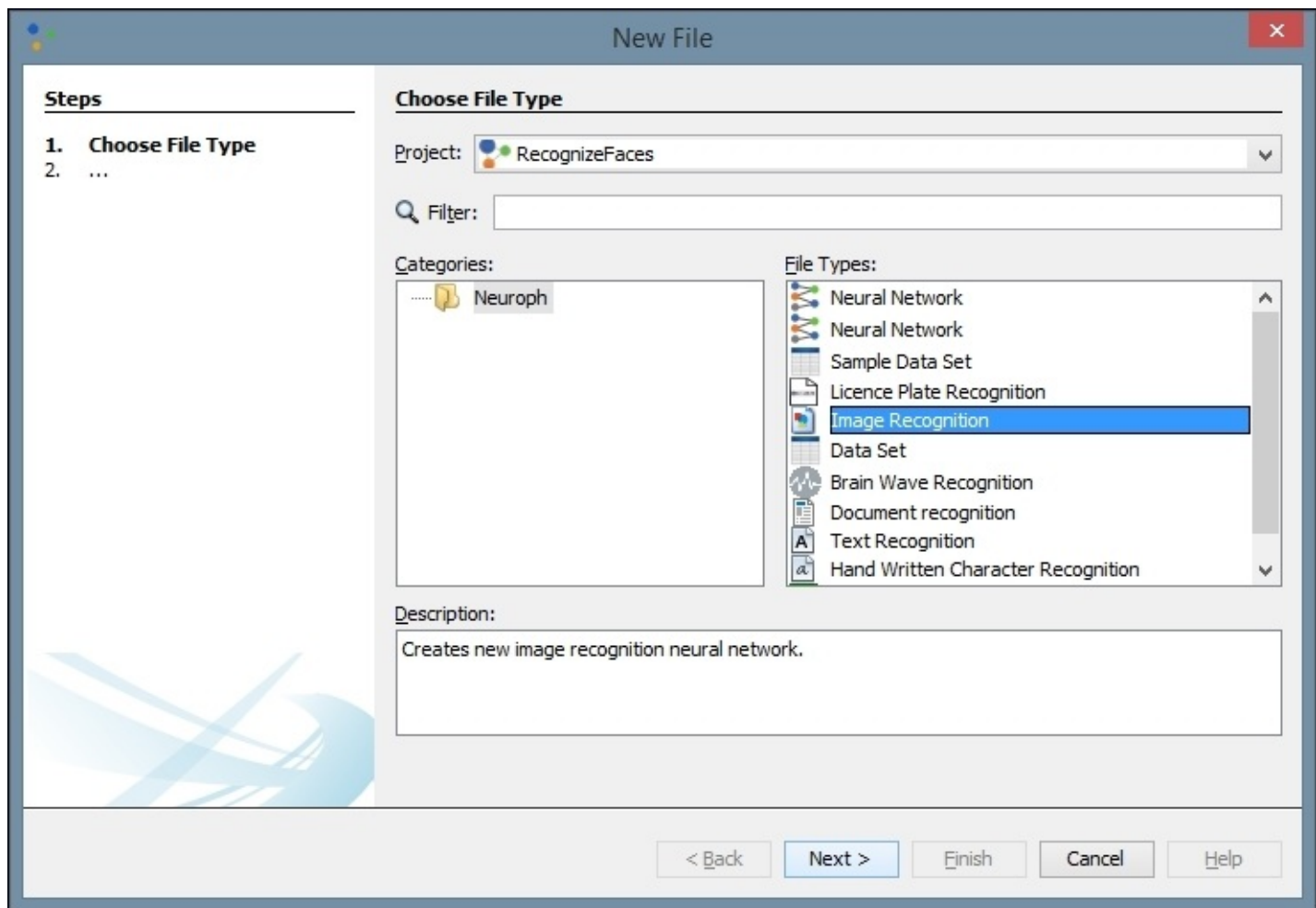
Click **Next** and then click **Add directory**. We have created a directory on our local machine and added several different black and white images of faces to use for training. These can be found by searching Google images or another search engine. The more quality images you have to train with, theoretically, the better your network will be:
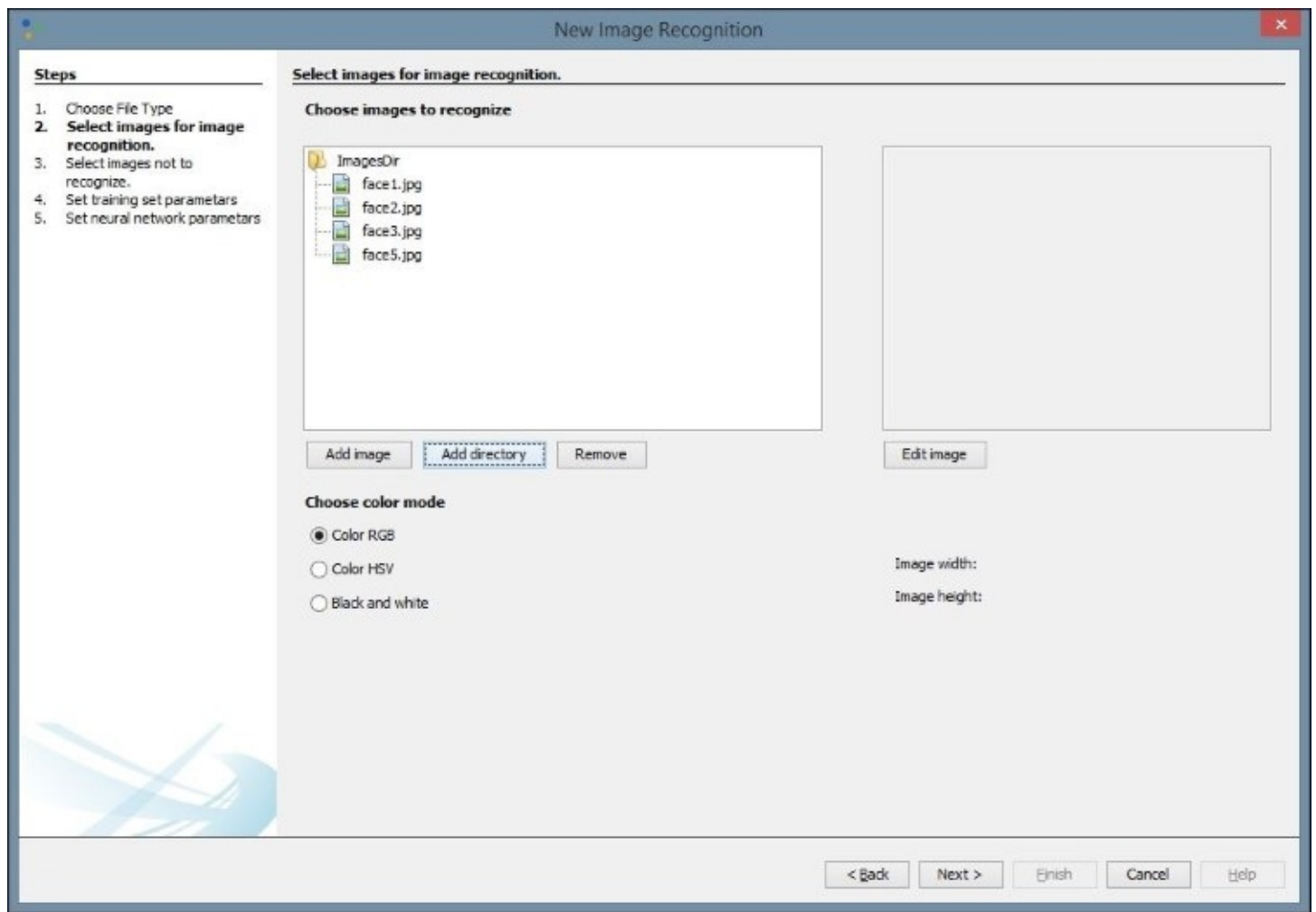
After you click **Next**, you will be directed to select an image to not recognize. You may need to try different images based upon the images you want to recognize. The image you select here will prevent false recognitions. We have chosen a simple blue square from another directory on our local machine, but if you are using other types of image for training, other color blocks may work better:
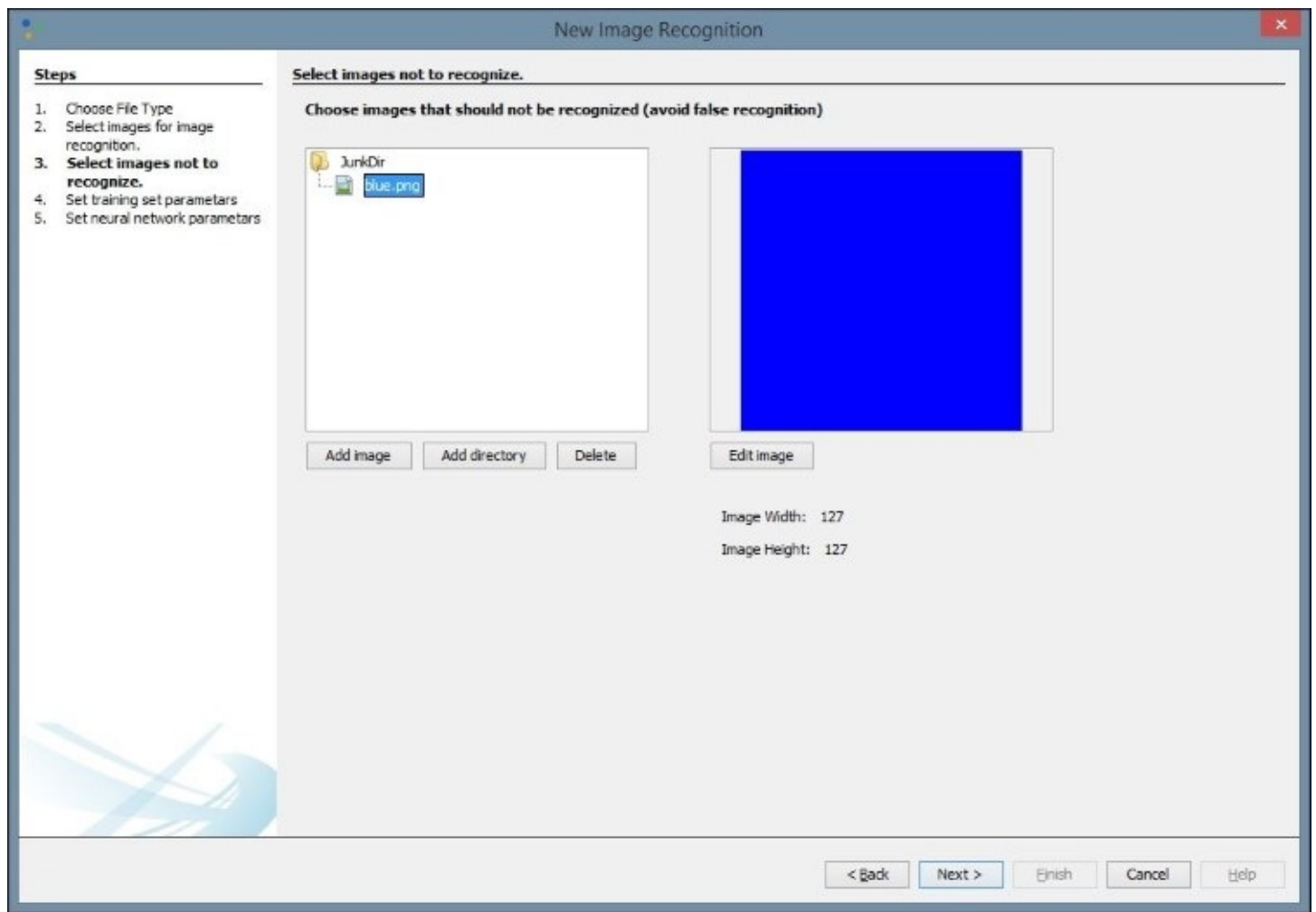
Next, we need to provide network training parameters. We also need to label our training dataset and set our resolution. A height and width of `20` is a good place to start, but you may want to change these values to improve your results. Some trial and error may be involved. The purpose of providing this information is to allow for image scaling. When we scale images to a smaller size, our network can process them and learn faster:
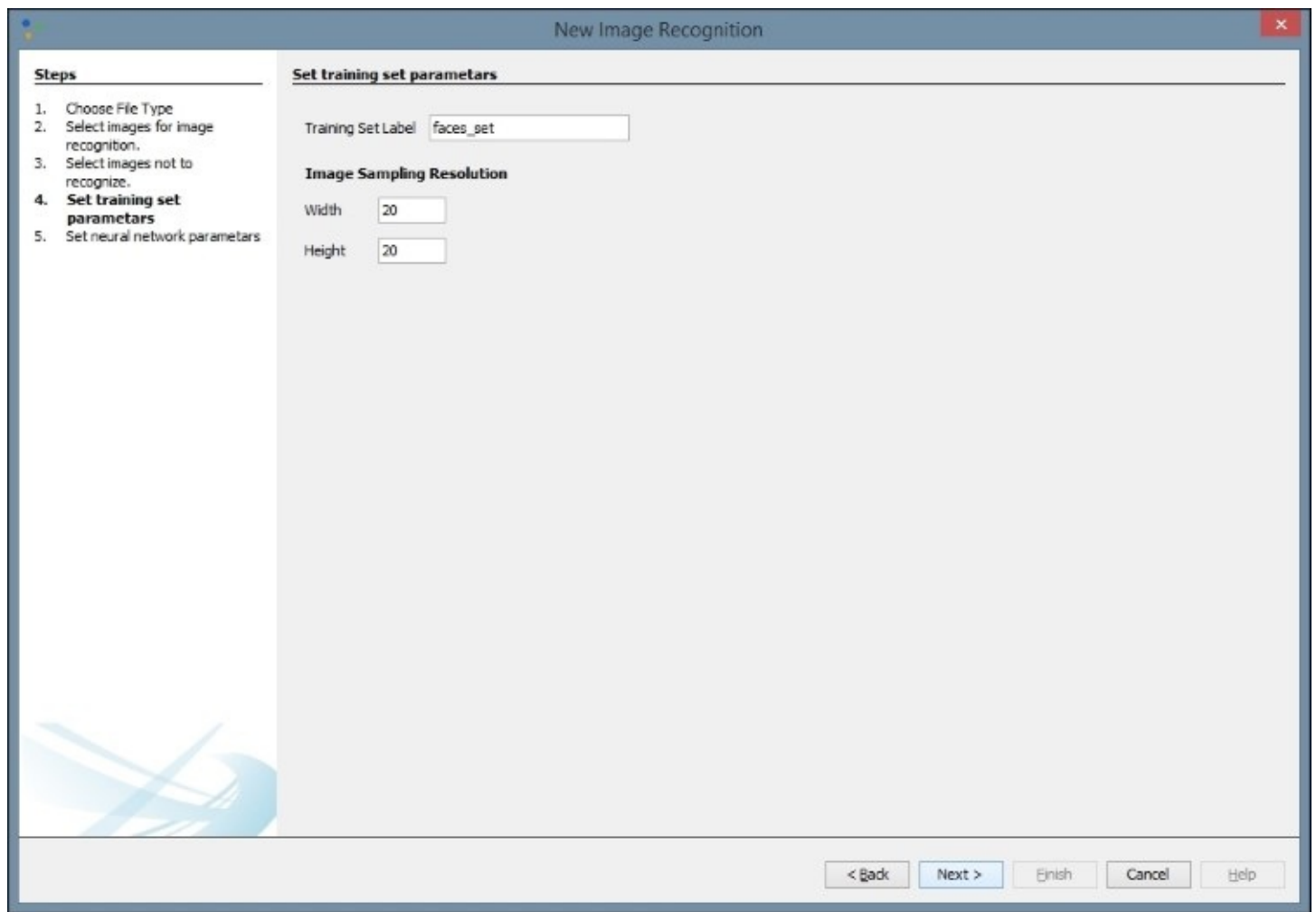
Finally, we can create our network. We assign a label to our network and define our **Transfer function**. The default function, **Sigmoid**, will work for most networks, but if your results are not optimal you may want to try **Tanh**. The default number of **Hidden Layers Neuron Counts** is 12, and that is a good place to start. Be aware that increasing the number of neurons increases the time it takes to train the network and decreases your ability to generalize the network to other images. As with some of our previous values, some trial and error may be necessary to find the optimal settings for a given network. Select **Finish** when you are done:
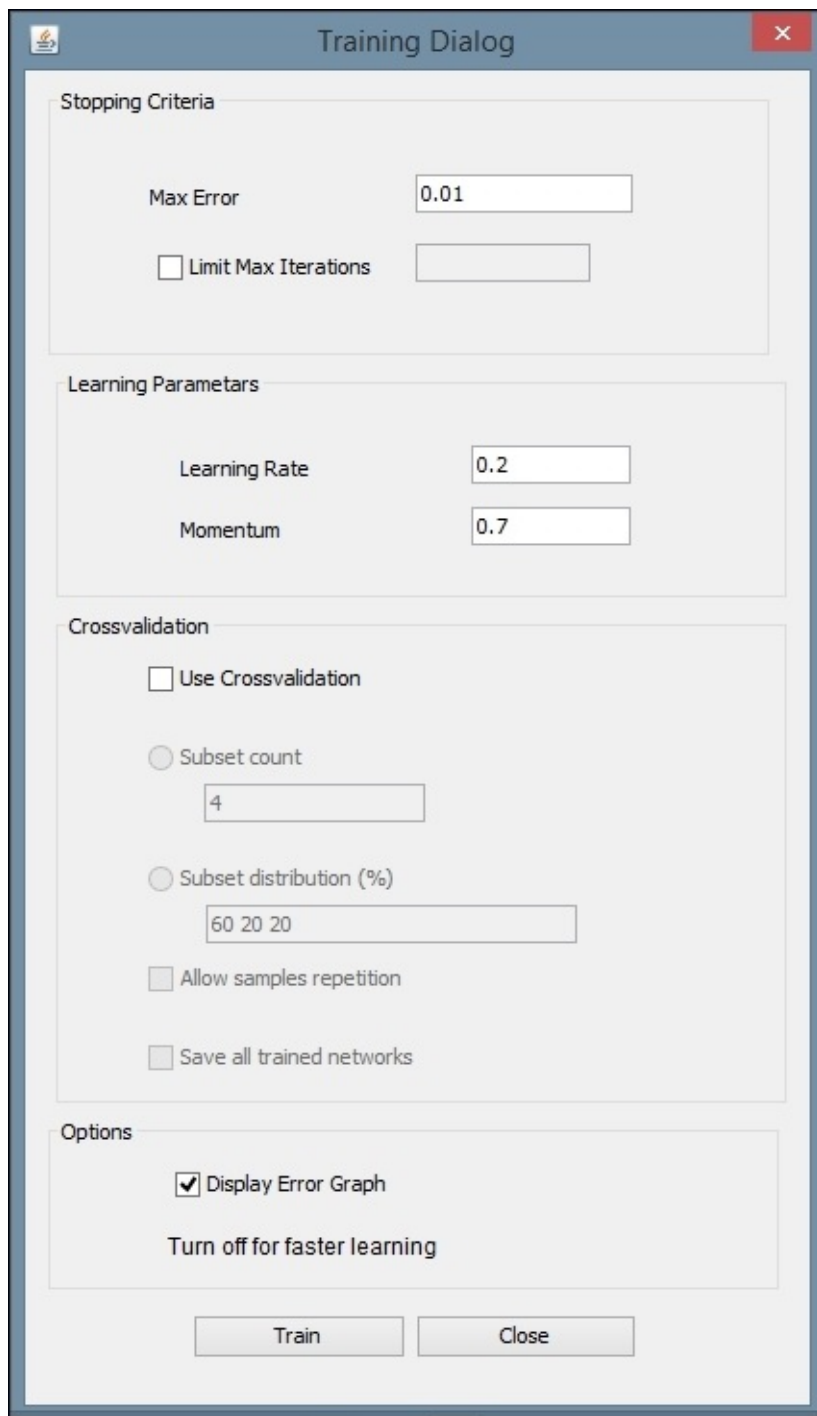
Training the model

Once we have created our network, we need to train it. Begin by double-clicking on your neural network in the left pane. This is the file with the `.nnet` extension. When you do this, you will open a visual representation of the network in the main window. Then drag the dataset, with the file extension `.tsest`, from the left pane to the top node of the neural network. You will notice the description on the node change to the name of your dataset. Next, click the **Train** button, located in the top-left part of the window:
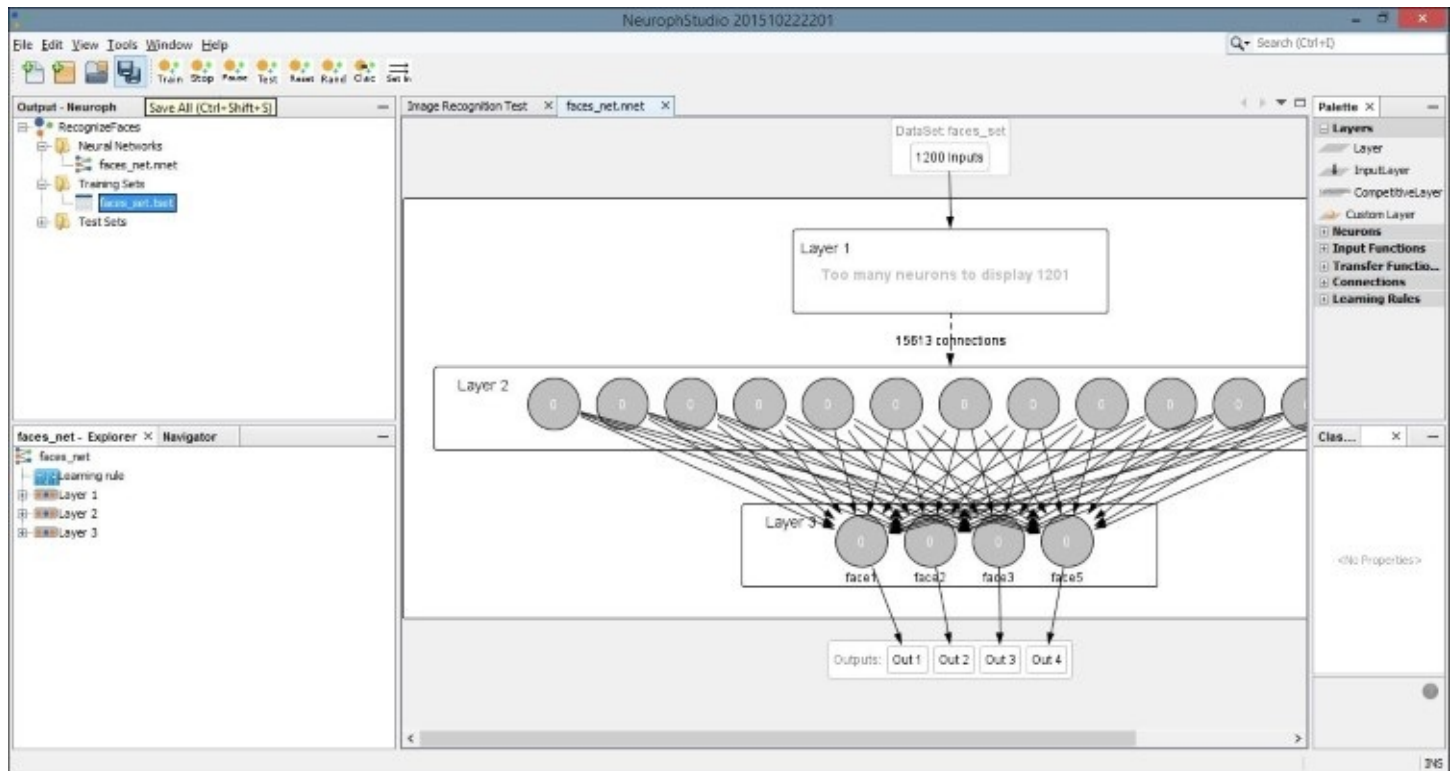
This will open a dialog box with settings for the training. You can leave the default values for **Max Error**, **Learning Rate**, and **Momentum**. Make sure the **Display Error Graph** box is checked. This allows you to see the improvement in the error rate as the training process continues:

After you click the **Train** button, you should see an error graph similar to the following one:
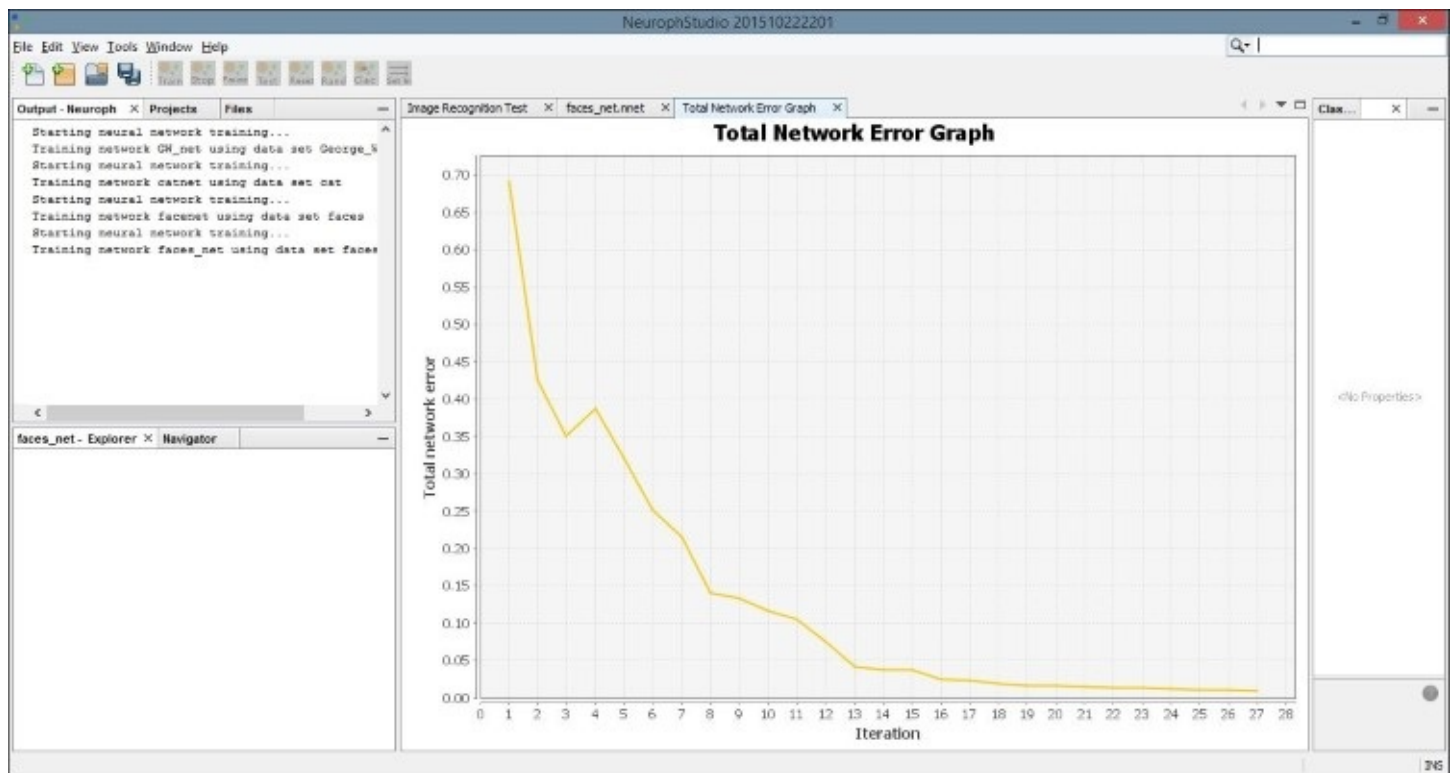
Select the tab titled **Image Recognition Test**. Then click on the **Select Test Image** button. We have loaded a simple image of a face that was not included in our original dataset:
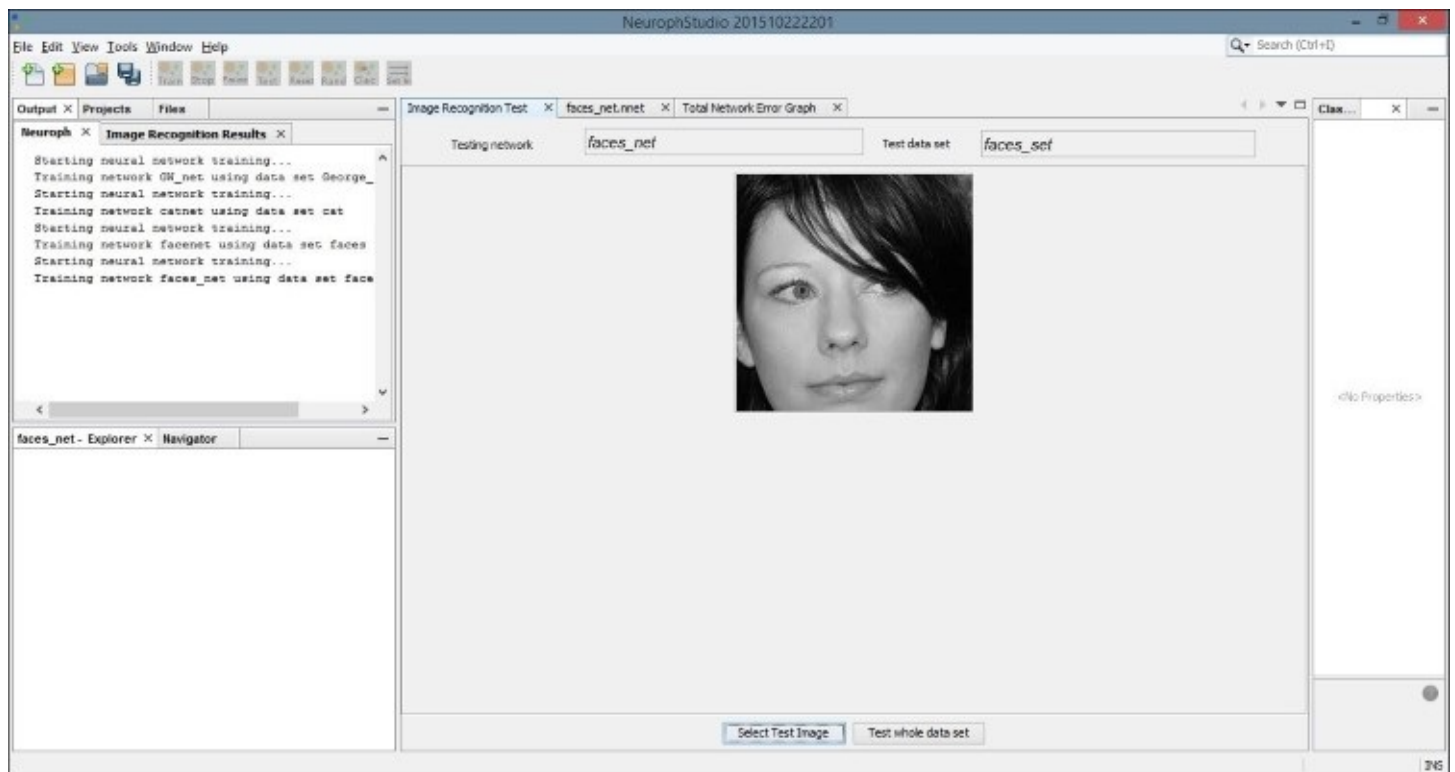
Locate the **Output** tab. It will be in the bottom or left pane and will display the results of comparing our test image with each image in the training set. The greater the number, the more closely our test image matches the image from our training set. The last image results in a greater output number than the first few comparisons. If we compare these images, they are more similar than the others in the dataset, and thus the network was able to create a more positive recognition of our test image:
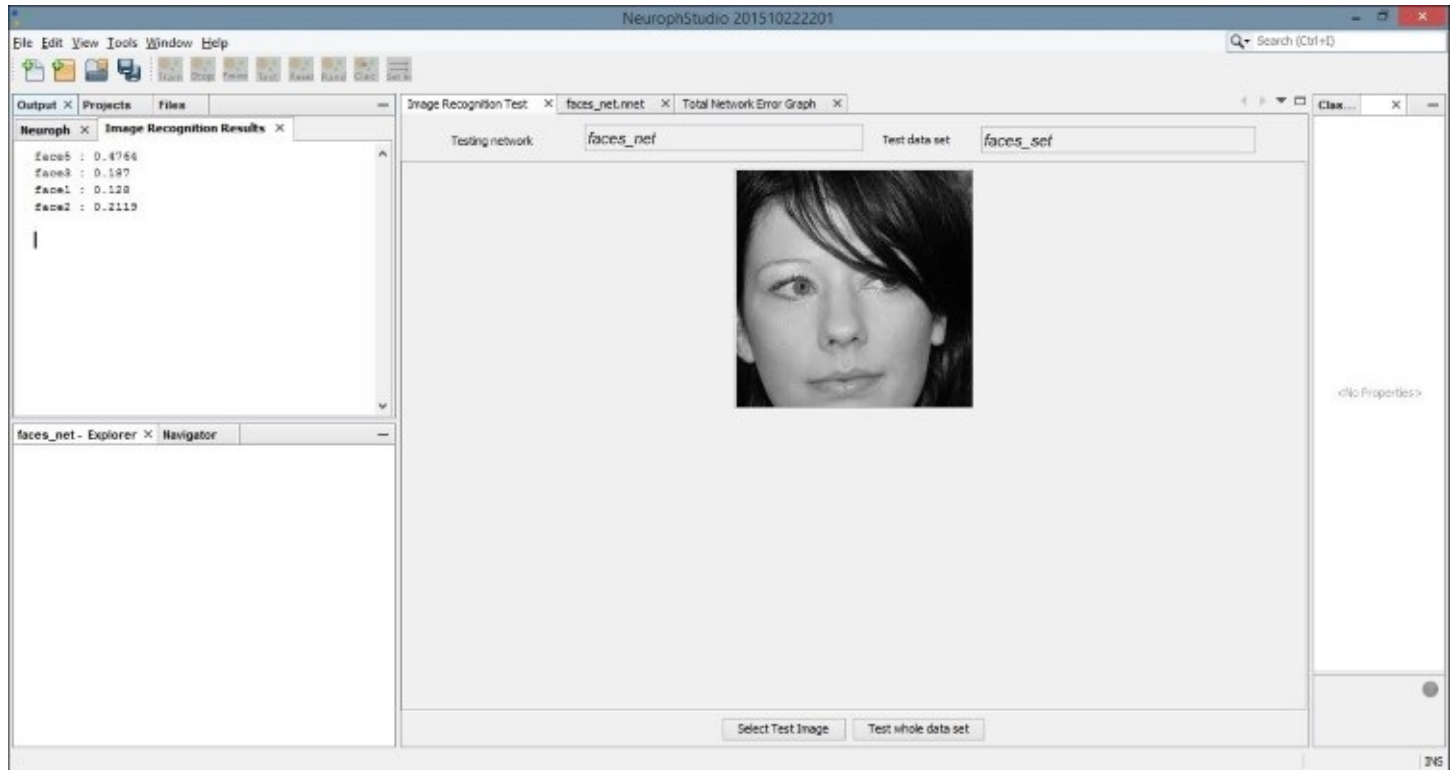
We can now save our network for later use. Select **Save** from the **File** menu and then you can use the `.nnet` file in external applications. The following code example shows a simple technique for running test data through your pre-built neural network. The `NeuralNetwork` class is part of the Neuroph core package, and the `load` method allows you to load the trained network into your project. Notice we used our neural network name, `faces_net`. We then retrieve the plugin for our image recognition file. Next, we call the `recognizeImage` method with a new image, which must handle an `IOException`. Our results are stored in a `HashMap` and printed to the console:

```
NeuralNetwork iRNet = NeuralNetwork.load("faces_net.nnet");
ImageRecognitionPlugin iRFile
  = (ImageRecognitionPlugin)iRNet.getPlugin(
    ImageRecognitionPlugin.class);
try {
    HashMap<String, Double> newFaceMap
      = imageRecognition.recognizeImage(
        new File("testFace.jpg"));
    out.println(newFaceMap.toString());
} catch(IOException e) {
   // Handle exceptions
}
```

This process allows us to use a GUI editor application to create our network in a more visual environment, but then embed the trained network into our own applications.

Summary

In this chapter, we demonstrated many techniques for processing speech and images. This capability is becoming important, as electronic devices are increasingly embracing these communication mediums.

TTS was demonstrated using FreeTSS. This technique allows a computer to present results as speech as opposed to text. We learned how we can control the attributes of the voice used, such as its gender and age.

Recognizing speech is useful and helps bridge the human-computer interface gap. We demonstrated how CMUSphinx is used to recognize human speech. As there is often more than one way speech can be interpreted, we learned how the API can return various options. We also demonstrated how individual words are extracted, along with the relative confidence that the right word was identified.

Image processing is a critical aspect of many applications. We started our discussion of image processing by use Tess4J to extract text from an image. This process is sometimes referred to as OCR. We learned, as with many visual and audio data files, that the quality of the results is related to the quality of the image.

We also learned how to use OpenCV to identify faces within an image. Information about specific view of faces, such as frontal or profile views, are contained in XML files. These files were used to outline faces within an image. More than one face can be detected at a time.

It can be helpful to classify images and sometimes external tools are useful for this purpose. We examined Neuroph Studio and created a neural network designed to recognize and classify images. We then tested our network with images of human faces.

In the next chapter, we will learn how to speed up common data science applications using multiple processors.

Chapter 11. Mathematical and Parallel Techniques for Data Analysis

The concurrent execution of a program can result in significant performance improvements. In this chapter, we will address the various techniques that can be used in data science applications. These can range from low-level mathematical calculations to higher-level API-specific options.

Always keep in mind that performance enhancement starts with ensuring that the correct set of application functionality is implemented. If the application does not do what a user expects, then the enhancements are for nought. The architecture of the application and the algorithms used are also more important than code enhancements. Always use the most efficient algorithm. Code enhancement should then be considered. We are not able to address the higher-level optimization issues in this chapter; instead, we will focus on code enhancements.

Many data science applications and supporting APIs use matrix operations to accomplish their tasks. Often these operations are buried within an API, but there are times when we may need to use these directly. Regardless, it can be beneficial to understand how these operations are supported. To this end, we will explain how matrix multiplication is handled using several different approaches.

Concurrent processing can be implemented using Java threads. A developer can use threads and thread pools to improve an application's response time. Many APIs will use threads when multiple CPUs or GPUs are not available, as is the case with **Aparapi**. We will not illustrate the use of threads here. However, the reader is assumed to have a basic knowledge of threads and thread pools.

The map-reduce algorithm is used extensively for data science applications. We will present a technique for achieving this type of parallel processing using Apache's Hadoop. Hadoop is a framework supporting the manipulation of large datasets, and can greatly decrease the required processing time for large data science projects. We will demonstrate a technique for calculating an average value for a sample set of data.

There are several well-known APIs that support multiple processors, including CUDA and OpenCL. CUDA is supported using **Java bindings for CUDA** (**JCuda**) (http://jcuda.org/). We will not demonstrate this technique directly here. However, many of the APIs we will use do support CUDA if it is available, such as DL4J. We will briefly discuss OpenCL and how it is supported in Java. It is worth nothing that the Aparapi API provides higher-level support, which may use either multiple CPUs or GPUs. A demonstration of Aparapi in support of matrix multiplication will be illustrated.

In this chapter, we will examine how multiple CPUs and GPUs can be harnessed to speed up data mining tasks. Many of the APIs we have used already take advantage of multiple processors or at least provide a means to enable GPU usage. We will introduce a number of these options in this

chapter.

Concurrent processing is also supported extensively in the cloud. Many of the techniques discussed here are used in the cloud. As a result, we will not explicitly address how to conduct parallel processing in the cloud.

Implementing basic matrix operations

There are several different types of matrix operations, including simple addition, subtraction, scalar multiplication, and various forms of multiplication. To illustrate the matrix operations, we will focus on what is known as **matrix product**. This is a common approach that involves the multiplication of two matrixes to produce a third matrix.

Consider two matrices, *A* and *B,* where matrix *A* has *n* rows and *m* columns. Matrix *B* will have *m* rows and *p* columns. The product of *A* and *B,* written as *AB,* is an *n* row and *p* column matrix. The *m* entries of the rows of *A* are multiplied by the *m* entries of the columns of matrix *B.* This is more explicitly shown here, where:

$$
A = \begin{vmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ A_{n1} & A_{n1} & \cdots & A_{nm} \end{vmatrix}
B = \begin{vmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ B_{m1} & B_{m1} & \cdots & B_{mp} \end{vmatrix}
$$

Where the product is defined as follows:

$$
(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}
$$

We start with the declaration and initialization of the matrices. The variables `n`, `m`, `p` represent the dimensions of the matrices. The `A` matrix is `n` by `m`, the `B` matrix is `m` by `p`, and the `C` matrix representing the product is `n` by `p`:

```
int n = 4;
int m = 2;
int p = 3;

double A[][] = {
```

```
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};
double B[][] = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
double C[][] = new double[n][p];
```

The following code sequence illustrates the multiplication operation using nested `for` loops:

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < m; k++) {
        for (int j = 0; j < p; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

This following code sequence formats output to display our matrix:

```
out.println("\nResult");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < p; j++) {
        out.printf("%.4f  ", C[i][j]);
    }
    out.println();
}
```

The result appears as follows:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Later, we will demonstrate several alternative techniques for performing the same operation. Next, we will discuss how to tailor support for multiple processors using DL4J.

Using GPUs with DeepLearning4j

DeepLearning4j works with GPUs such as those provided by NVIDIA. There are options available that enable the use of GPUs, specify how many GPUs should be used, and control the use of GPU memory. In this section, we will show how you can use these options. This type of control is often available with other high-level APIs.

DL4J uses **n-dimensional arrays for Java** (**ND4J**) (http://nd4j.org/) to perform numerical computations. This is a library that supports n-dimensional array objects and other numerical computations, such as linear algebra and signal processing. It includes support for GPUs and is also integrated with Hadoop and Spark.

A **vector** is a one-dimensional array of numbers and is used extensively with neural networks. A vector is a type of mathematical structure called a **tensor**. A tensor is essentially a multidimensional array. We can think of a tensor as an array with three or more dimensions, and each dimension is called a **rank**.

There is often a need to map a multidimensional set of numbers to a one-dimensional array. This is done by flattening the array using a defined order. For example, with a two-dimensional array many systems will allocate the members of the array in row-column order. This means the first row is added to the vector, followed by the second vector, and then the third, and so forth. We will use this approach in the *Using the ND4J API* section.

To enable GPU use, the project's POM file needs to be modified. In the properties section of the POM file, the `nd4j.backend` tag needs to added or modified, as shown here:

```
<nd4j.backend>nd4j-cuda-7.5-platform</<nd4j.backend>
```

Models can be trained in parallel using the `ParallelWrapper` class. The training task is automatically distributed among available CPUs/GPUs. The model is used as an argument to the `ParallelWrapper` class' `Builder` constructor, as shown here:

```
ParallelWrapper parallelWrapper =
    new ParallelWrapper.Builder(aModel)
        // Builder methods...
        .build();
```

When executed, a copy of the model is used on each GPU. After the number of iterations is specified by the `averagingFrequency` method, the models are averaged and then the training process continues.

There are various methods that can be used to configure the class, as summarized in the following table:

Method	Purpose

prefetchBuffer	Specifies the size of a buffer used to pre-fetch data
workers	Specifies the number of workers to be used
averageUpdaters averagingFrequency reportScoreAfterAveraging useLegacyAveraging	Various methods to control how averaging is achieved

The number of workers should be greater than the number of GPUs available.

As with most computations, using a lower precision value will speed up the processing. This can be controlled using the `setDTypeForContext` method, as shown next. In this case, half precision is specified:

```
DataTypeUtil.setDTypeForContext(DataBuffer.Type.HALF);
```

This support and more details regarding optimization techniques can be found at http://deeplearning4j.org/gpu.

Using map-reduce

Map-reduce is a model for processing large sets of data in a parallel, distributed manner. This model consists of a `map` method for filtering and sorting data, and a `reduce` method for summarizing data. The map-reduce framework is effective because it distributes the processing of a dataset across multiple servers, performing mapping and reduction simultaneously on smaller pieces of the data. Map-reduce provides significant performance improvements when implemented in a multi-threaded manner. In this section, we will demonstrate a technique using Apache's Hadoop implementation. In the *Using Java 8 to perform map-reduce* section, we will discuss techniques for performing map-reduce using Java 8 streams.

Hadoop is a software ecosystem providing support for parallel computing. Map-reduce jobs can be run on Hadoop servers, generally set up as clusters, to significantly improve processing speeds. Hadoop has trackers that run map-reduce operations on nodes within a Hadoop cluster. Each node operates independently and the trackers monitor the progress and integrate the output of each node to generate the final output. The following image can be found at http://www.developer.com/java/data/big-data-tool-map-reduce.html and demonstrates the basic map-reduce model with trackers.

Using Apache's Hadoop to perform map-reduce

We are going to show you a very simple example of a map-reduce application here. Before we can use Hadoop, we need to download and extract Hadoop application files. The latest versions can be found at http://hadoop.apache.org/releases.html. We are using version 2.7.3 for this demonstration.

You will need to set your `JAVA_HOME` environment variable. Additionally, Hadoop is intolerant of long file paths and spaces within paths, so make sure you extract Hadoop to the simplest directory structure possible.

We will be working with a sample text file containing information about books. Each line of our tab-delimited file has the book title, author, and page count:

```
Moby Dick  Herman Melville  822

Charlotte's Web  E.B. White  189
The Grapes of Wrath  John Steinbeck  212
Jane Eyre  Charlotte Bronte  299
A Tale of Two Cities  Charles Dickens  673
War and Peace  Leo Tolstoy  1032
The Great Gatsby  F. Scott Fitzgerald  275
```

We are going to use a `map` function to extract the title and page count information and then a `reduce` function to calculate the average page count of the books in our dataset. To begin, create a new class, `AveragePageCount`. We will create two static classes within `AveragePageCount`, one to handle the map procedure and one to handle the reduction.

Writing the map method

First, we will create the `TextMapper` class, which will implement the `map` method. This class inherits from the `Mapper` class and has two private instance variables, `pages` and `bookTitle`. `pages` is an `IntWritable` object and `bookTitle` is a `Text` object. `IntWritable` and `Text` are used because these objects will need to be serialized to a byte stream before they can be transmitted to the servers for processing. These objects take up less space and transfer faster than the comparable `int` or `String` objects:

```
public static class TextMapper
        extends Mapper<Object, Text, Text, IntWritable> {

    private final IntWritable pages = new IntWritable();
    private final Text bookTitle = new Text();

}
```

Within our `TextMapper` class we create the `map` method. This method takes three parameters: the `key` object, a `Text` object, `bookInfo`, and the `Context`. The key allows the tracker to map each particular object back to the correct job. The `bookInfo` object contains the text or string data about each book. `Context` holds information about the entire system and allows the method to report on progress and update values within the system.

Within the `map` method, we use the `split` method to break each piece of book information into an array of `String` objects. We set our `bookTitle` variable to position `0` of the array and set `pages` to the value stored in position `2`, after parsing it as an integer. We can then write out our book title and page count information through the context and update our entire system:

```
public void map(Object key, Text bookInfo, Context context)
 throws IOException, InterruptedException {
        String[] book = bookInfo.toString().split("\t");
        bookTitle.set(book[0]);
        pages.set(Integer.parseInt(book[2]));
        context.write(bookTitle, pages);
    }
```

Writing the reduce method

Next, we will write our `AverageReduce` class. This class extends the `Reducer` class and will perform the reduction processes to calculate our average page count. We have created four variables for this class: a `FloatWritable` object to store our average page count, a float `average` to hold our temporary average, a float `count` to count how many books exist in our dataset, and an integer `sum` to add up the page counts:

```
public static class AverageReduce
        extends Reducer<Text, IntWritable, Text, FloatWritable> {

    private final FloatWritable finalAvg = new FloatWritable();
    Float average = 0f;
    Float count = 0f;
    int sum = 0;

}
```

Within our `AverageReduce` class we will create the `reduce` method. This method takes as input a `Text` key, an `Iterable` object holding writeable integers representing the page counts, and the `Context`. We use our iterator to process the page counts and add each to our sum. We then calculate the average and set the value of `finalAvg`. This information is paired with a `Text` object label and written to the `Context`:

```
public void reduce(Text key, Iterable<IntWritable> pageCnts,
    Context context)
        throws IOException, InterruptedException {

for (IntWritable cnt : pageCnts) {
    sum += cnt.get();
}
count += 1;
average = sum / count;
finalAvg.set(average);
context.write(new Text("Average Page Count = "), finalAvg);
}
```

Creating and executing a new Hadoop job

We are now ready to create our `main` method in the same class and execute our map-reduce processes. To do this, we need to create a new `Configuration` object and a new `Job`. We then set up the significant classes to use in our application.

```
public static void main(String[] args) throws Exception {
    Configuration con = new Configuration();
    Job bookJob = Job.getInstance(con, "Average Page Count");
    ...
}
```

We set our main class, `AveragePageCount`, in the `setJarByClass` method. We specify our `TextMapper` and `AverageReduce` classes using the `setMapperClass` and `setReducerClass` methods, respectively. We also specify that our output will have a text-based key and a writeable integer using the `setOutputKeyClass` and `setOutputValueClass` methods:

```
    bookJob.setJarByClass(AveragePageCount.class);
    bookJob.setMapperClass(TextMapper.class);
    bookJob.setReducerClass(AverageReduce.class);
    bookJob.setOutputKeyClass(Text.class);
    bookJob.setOutputValueClass(IntWritable.class);
```

Finally, we create new input and output paths using the `addInputPath` and `setOutputPath` methods. These methods both take our `Job` object as the first parameter and a `Path` object representing our input and output file locations as the second parameter. We then call `waitForCompletion`. Our application exits once this call returns true:

```
    FileInputFormat.addInputPath(bookJob, new Path("C:/Hadoop/books.txt"));
    FileOutputFormat.setOutputPath(bookJob, new
        Path("C:/Hadoop/BookOutput"));
    if (bookJob.waitForCompletion(true)) {
        System.exit(0);
    }
```

To execute the application, open a command prompt and navigate to the directory containing our `AveragePageCount.class` file. We then use the following command to execute our sample application:

```
hadoop AveragePageCount
```

While our task is running, we see updated information about our process output to the screen. A sample of our output is shown as follows:

```
...
File System Counters
    FILE: Number of bytes read=1132
    FILE: Number of bytes written=569686
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
```

```
Map-Reduce Framework
    Map input records=7
    Map output records=7
    Map output bytes=136
    Map output materialized bytes=156
    Input split bytes=90
    Combine input records=0
    Combine output records=0
    Reduce input groups=7
    Reduce shuffle bytes=156
    Reduce input records=7
    Reduce output records=7
    Spilled Records=14
    Shuffled Maps =1
    Failed Shuffles=0
    Merged Map outputs=1
    GC time elapsed (ms)=11
    Total committed heap usage (bytes)=536870912
Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
File Input Format Counters
    Bytes Read=249
File Output Format Counters
     Bytes Written=216
```

If we open the `BookOutput` directory created on our local machine, we find four new files. Use a text editor to open `part-r-00000`. This file contains information about the average page count as it was calculated using parallel processes. A sample of this output follows:

```
Average Page Count =  673.0
Average Page Count =   431.0
Average Page Count =   387.0
Average Page Count =   495.75
Average Page Count =   439.0
Average Page Count =   411.66666
Average Page Count =   500.2857
```

Notice how the average changes as each individual process is combined with the other reduction processes. This has the same effect as calculating the average of the first two books first, then adding in the third book, then the fourth, and so on. The advantage here of course is that the averaging is done in a parallel manner. If we had a huge dataset, we should expect to see a noticeable advantage in execution time. The last line of `BookOutput` reflects the correct and final average of all seven page counts.

Various mathematical libraries

There are numerous mathematical libraries available for Java use. In this section, we will provide a quick and high-level overview of several libraries. These libraries do not necessarily automatically support multiple processors. In addition, the intent of this section is to provide some insight into how these libraries can be used. In most cases, they are relatively easy to use.

A list of Java mathematical libraries is found at https://en.wikipedia.org/wiki/List_of_numerical_libraries#Java and https://java-matrix.org/. We will demonstrate the use of the jblas, Apache Commons Math, and the ND4J libraries.

Using the jblas API

The jblas API (http://jblas.org/) is a math library supporting Java. It is based on **Basic Linear Algebra Subprograms** (**BLAS**) (http://www.netlib.org/blas/) and **Linear Algebra Package** (**LAPACK**) (http://www.netlib.org/lapack/), which are standard libraries for fast arithmetic calculation. The jblas API provides a wrapper around these libraries.

The following is a demonstration of how matrix multiplication is performed. We start with the matrix definitions:

```
DoubleMatrix A = new DoubleMatrix(new double[][]{
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}});

DoubleMatrix B = new DoubleMatrix(new double[][]{
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}});
DoubleMatrix C;
```

The actual statement to perform multiplication is quite short, as shown next. The `mmul` method is executed against the `A` matrix, where the `B` array is passed as an argument:

```
C = A.mmul(B);
```

The resulting `C` matrix is then displayed:

```
for(int i=0; i<C.getRows(); i++) {
    out.println(C.getRow(i));
}
```

The output should be as follows:

```
[0.027616, 0.199927, 0.213192]
[0.144288, 0.411798, 0.541650]
[0.348579, 0.328344, 0.705819]
[0.196399, 0.294114, 0.496353]
```

This library is fairly easy to use and supports an extensive set of arithmetic operations.

Using the Apache Commons math API

The Apache Commons math API (http://commons.apache.org/proper/commons-math/) supports a large number of mathematical and statistical operations. The following example illustrates how to perform matrix multiplication.

We start with the declaration and initialization of the `A` and `B` matrices:

```
double[][] A = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};

double[][] B = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
```

Apache Commons uses the `RealMatrix` class to hold a matrix. In the following code sequence, the corresponding matrices for the `A` and `B` matrices are created using the `Array2DRowRealMatrix` constructor:

```
RealMatrix aRealMatrix = new Array2DRowRealMatrix(A);
RealMatrix bRealMatrix = new Array2DRowRealMatrix(B);
```

The multiplication is straightforward using the multiply method, as shown next:

```
RealMatrix cRealMatrix = aRealMatrix.multiply(bRealMatrix);
```

The next `for` loop will display the following results:

```
for (int i = 0; i < cRealMatrix.getRowDimension(); i++) {
    out.println(cRealMatrix.getRowVector(i));
}
```

The output should be as follows:

```
{0.02761552; 0.19992684; 0.2131916}
{0.14428772; 0.41179806; 0.54165016}
{0.34857924; 0.32834382; 0.70581912}
{0.19639854; 0.29411363; 0.4963528}
```

Using the ND4J API

ND4J (http://nd4j.org/) is the library used by DL4J to perform arithmetic operations. The library is also available for direct use. In this section, we will demonstrate how matrix multiplication is performed using the A and B matrices.

Before we can perform the multiplication, we need to flatten the matrices to vectors. The following declares and initializes these vectors:

```
double[] A = {
    0.1950, 0.0311,
    0.3588, 0.2203,
    0.1716, 0.5931,
    0.2105, 0.3242};

double[] B = {
    0.0502, 0.9823, 0.9472,
    0.5732, 0.2694, 0.916};
```

The Nd4j class' create method creates an INDArray instance given a vector and dimension information. The first argument of the method is the vector. The second argument specifies the dimensions of the matrix. The last argument specifies the order the rows and columns are laid out. This order is either row-column major as exemplified by c, or column-row major order as used by FORTRAN. Row-column order means the first row is allocated to the vector, followed by the second row, and so forth.

In the following code sequence 2INDArray instances are created using the A and B vectors. The first is a 4 row, 2 column matrix using row-major order as specified by the third argument, c. The second INDArray instance represents the B matrix. If we wanted to use column-row ordering, we would use an f instead.

```
INDArray aINDArray = Nd4j.create(A,new int[]{4,2},'c');
INDArray bINDArray = Nd4j.create(B,new int[]{2,3},'c');
```

The C array, represented by cINDArray, is then declared and assigned the result of the multiplication. The mmul performs the operation:

```
INDArray cINDArray;
cINDArray = aINDArray.mmul(bINDArray);
```

The following sequence displays the results using the getRow method:

```
for(int i=0; i<cINDArray.rows(); i++) {
   out.println(cINDArray.getRow(i));
}
```

The output should be as follows:

```
[0.03, 0.20, 0.21]
[0.14, 0.41, 0.54]
```

```
[0.35, 0.33, 0.71]
[0.20, 0.29, 0.50]
```

Next, we will provide an overview of the OpenCL API that provide supports for concurrent operations on a number of platforms.

Using OpenCL

Open Computing Language (**OpenCL**) (https://www.khronos.org/opencl/) supports programs that execute across heterogeneous platforms, that is, platforms potentially using different vendors and architectures. The platforms can use different processing units, including **Central Processing Unit** (**CPU**), **Graphical Processing Unit** (**GPU**), **Digital Signal Processor** (**DSP**), **Field-Programmable Gate Array** (**FPGA**), and other types of processors.

OpenCL uses a C99-based language to program the devices, providing a standard interface for programming concurrent behavior. OpenCL supports an API that allows code to be written in different languages. For Java, there are several APIs that support the development of OpenCL based languages:

- **Java bindings for OpenCL** (**JOCL**) (http://www.jocl.org/) - This is a binding to the original OpenCL C implementation and can be verbose.
- **JavaCl** (https://code.google.com/archive/p/javacl/) - Provides an object-oriented interface to JOCL.
- **Java OpenCL** (http://jogamp.org/jocl/www/) - Also provides an object-oriented abstraction of JOCL. It is not intended for client use.
- The **Lightweight Java Game Library** (**LWJGL**) (https://www.lwjgl.org/) - Also provides support for OpenCL and is oriented toward GUI applications.

In addition, Aparapi provides higher-level access to OpenCL, thus avoiding some of the complexity involved in creating OpenCL applications.

Code that runs on a processor is encapsulated in a kernel. Multiple kernels will execute in parallel on different computing devices. There are different levels of memory supported by OpenCL. A specific device may not support each level. The levels include:

- **Global memory** - Shared by all computing units
- **Read-only memory** - Generally not writable
- **Local memory** - Shared by a group of computing units
- **Per-element private memory** - Often a register

OpenCL applications require a considerable amount of initial code to be useful. This complexity does not permit us to provide a detailed example of its use. However, the Aparapi section does provide some feel for how OpenCL applications are structured.

Using Aparapi

Aparapi (https://github.com/aparapi/aparapi) is a Java library that supports concurrent operations. The API supports code running on GPUs or CPUs. GPU operations are executed using OpenCL, while CPU operations use Java threads. The user can specify which computing resource to use. However, if GPU support is not available, Aparapi will revert to Java threads.

The API will convert Java byte codes to OpenCL at runtime. This makes the API largely independent from the graphics card used. The API was initially developed by AMD but has been released as open source. This is reflected in the basic package name, `com.amd.aparari`. Aparapi offers a higher level of abstraction than provided by OpenCL.

Aparapi code is located in a class derived from the `Kernel` class. Its `execute` method will start the operations. This will result in an internal call to a `run` method, which needs to be overridden. It is within the `run` method that concurrent code is placed. The `run` method is executed multiple times on different processors.

Due to OpenCL limitations, we are unable to use inheritance or method overloading. In addition, it does not like `println` in the `run` method, since the code may be running on a GPI. Aparapi only supports one-dimensional arrays. Arrays using two or more dimensions need to be flattened to a one dimension array. The support for double values is dependent on the OpenCL version and GPU configuration.

When a Java thread pool is used, it allocates one thread per CPU core. The kernel containing the Java code is cloned, one copy per thread. This avoids the need to access data across a thread. Each thread has access to information, such as a global ID, to assist in the code execution. The kernel will wait for all of the threads to complete.

Aparapi downloads can be found at https://github.com/aparapi/aparapi/releases.

Creating an Aparapi application

The basic framework for an Aparapi application is shown next. It consists of a `Kernel` derived class where the `run` method is overridden. In this example, the `run` method will perform scalar multiplication. This operation involves multiplying each element of a vector by some value.

The `ScalarMultiplicationKernel` extends the `Kernel` class. It possesses two instance variables used to hold the matrices for input and output. The constructor will initialize the matrices. The `run` method will perform the actual computations, and the `displayResult` method will show the results of the multiplication:

```
public class ScalarMultiplicationKernel extends Kernel {
    float[] inputMatrix;
    float outputMatrix [];

    public ScalarMultiplicationKernel(float inputMatrix[]) {
        ...
    }

    @Override
    public void run() {
        ...
    }

    public void displayResult() {
        ...
    }
}
```

The constructor is shown here:

```
public ScalarMultiplicationKernel(float inputMatrix[]) {
    this.inputMatrix = inputMatrix;
    outputMatrix = new float[this.inputMatrix.length];
}
```

In the `run` method, we use a global ID to index into the matrix. This code is executed on each computation unit, for example, a GPU or thread. A unique global ID is provided to each computational unit, allowing the code to access a specific element of the matrix. In this example, each element of the input matrix is multiplied by 2 and then assigned to the corresponding element of the output matrix:

```
public void run() {
    int globalID = this.getGlobalId();
    outputMatrix[globalID] = 2.0f * inputMatrix[globalID];
}
```

The `displayResult` method simply displays the contents of the `outputMatrix` array:

```
public void displayResult() {
    out.println("Result");
    for (float element : outputMatrix) {
```

```
        out.printf("%.4f ", element);
    }
    out.println();
}
```

To use this kernel, we need to declare variables for the `inputMatrix` and its `size`. The `size` will be used to control how many kernels to execute:

```
float inputMatrix[] = {3, 4, 5, 6, 7, 8, 9};
int size = inputMatrix.length;
```

The kernel is then created using the input matrix followed by the invocation of the `execute` method. This method starts the process and will eventually invoke the `Kernel` class' `run` method based on the `execute` method's argument. This argument is referred to as the pass ID. While not used in this example, we will use it in the next section. When the process is complete, the resulting output matrix is displayed and the `dispose` method is called to stop the process:

```
ScalarMultiplicationKernel kernel =
        new ScalarMultiplicationKernel(inputMatrix);
kernel.execute(size);
kernel.displayResult();
kernel.dispose();
```

When this application is executed we will get the following output:

6.0000 8.0000 10.0000 12.0000 14.0000 16.0000 18.000

We can specify the execution mode using the Kernel class' `setExecutionMode` method, as shown here:

```
kernel.setExecutionMode(Kernel.EXECUTION_MODE.GPU);
```

However, it is best to let Aparapi determine the execution mode. The following table summarizes the execution modes available:

Execution mode	Meaning
`Kernel.EXECUTION_MODE.NONE`	Does not specify mode
`Kernel.EXECUTION_MODE.CPU`	Use CPU
`Kernel.EXECUTION_MODE.GPU`	Use GPU
`Kernel.EXECUTION_MODE.JTP`	Use Java threads
`Kernel.EXECUTION_MODE.SEQ`	

	Use single loop (for debugging purposes)

Next, we will demonstrate how we can use Aparapi to perform dot product matrix multiplication.

Using Aparapi for matrix multiplication

We will use the matrices as used in the *Implementing basic matrix operations* section. We start with the declaration of the `MatrixMultiplicationKernel` class, which contains the vector declarations, a constructor, the `run` method, and a `displayResults` method. The vectors for matrices `A` and `B` have been flattened to one-dimensional arrays by allocating the matrices in row-column order:

```
class MatrixMultiplicationKernel extends Kernel {
    float[] vectorA = {
        0.1950f, 0.0311f, 0.3588f,
        0.2203f, 0.1716f, 0.5931f,
        0.2105f, 0.3242f};
    float[] vectorB = {
        0.0502f, 0.9823f, 0.9472f,
        0.5732f, 0.2694f, 0.916f};
    float[] vectorC;
    int n;
    int m;
    int p;

    @Override
    public void run() {
        ...
    }

    public MatrixMultiplicationKernel(int n, int m, int p) {
        ...
    }

    public void displayResults () {
        ...
    }
}
```

The `MatrixMultiplicationKernel` constructor assigns values for the matrices' dimensions and allocates memory for the result stored in `vectorC`, as shown here:

```
public MatrixMultiplicationKernel(int n, int m, int p) {
    this.n = n;
    this.p = p;
    this.m = m;
    vectorC = new float[n * p];
}
```

The run method uses a global ID and a pass ID to perform the matrix multiplication. The pass ID is specified as the second argument of the `Kernel` class' `execute` method, as we will see shortly. This value allows us to advance the column index for `vectorC`. The vector indexes map to the corresponding row and column positions of the original matrices:

```
public void run() {
    int i = getGlobalId();
    int j = this.getPassId();
```

```
        float value = 0;
        for (int k = 0; k < p; k++) {
            value += vectorA[k + i * m] * vectorB[k * p + j];
        }
        vectorC[i * p + j] = value;
}
```

The `displayResults` method is shown as follows:

```
public void displayResults() {
    out.println("Result");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            out.printf("%.4f  ", vectorC[i * p + j]);
        }
        out.println();
    }
}
```

The kernel is started in the same way as in the previous section. The `execute` method is passed the number of kernels that should be created and an integer indicating the number of passes to make. The number of passes is used to control the index into the `vectorA` and `vectorB` arrays:

```
MatrixMultiplicationKernel kernel = new MatrixMultiplicationKernel(n, m,
    p);kernel.execute(6, 3);kernel.displayResults();
kernel.dispose();
```

When this example is executed, you will get the following output:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Next, we will see how Java 8 additions can contribute to solving math-intensive problems in a parallel manner.

Using Java 8 streams

The release of Java 8 came with a number of important enhancements to the language. The two enhancements of interest to us include lambda expressions and streams. A lambda expression is essentially an anonymous function that adds a functional programming dimension to Java. The concept of streams, as introduced in Java 8, does not refer to IO streams. Instead, you can think of it as a sequence of objects that can be generated and manipulated using a fluent style of programming. This style will be demonstrated shortly.

As with most APIs, programmers must be careful to consider the actual execution performance of their code using realistic test cases and environments. If not used properly, streams may not actually provide performance improvements. In particular, parallel streams, if not crafted carefully, can produce incorrect results.

We will start with a quick introduction to lambda expressions and streams. If you are familiar with these concepts you may want to skip over the next section.

Understanding Java 8 lambda expressions and streams

A lambda expression can be expressed in several different forms. The following illustrates a simple lambda expression where the symbol, ->, is the lambda operator. This will take some value, e, and return the value multiplied by two. There is nothing special about the name e. Any valid Java variable name can be used:

```
e -> 2 * e
```

It can also be expressed in other forms, such as the following:

```
(int e) -> 2 * e
(double e) -> 2 * e
(int e) -> {return 2 * e;
```

The form used depends on the intended value of e. Lambda expressions are frequently used as arguments to a method, as we will see shortly.

A stream can be created using a number of techniques. In the following example, a stream is created from an array. The IntStream interface is a type of stream that uses integers. The Arrays class' stream method converts an array into a stream:

```
IntStream stream = Arrays.stream(numbers);
```

We can then apply various stream methods to perform an operation. In the following statement, the forEach method will simply display each integer in the stream:

```
stream.forEach(e -> out.printf("%d  ", e));
```

There are a variety of stream methods that can be applied to a stream. In the following example, the mapToDouble method will take an integer, multiply it by 2, and then return it as a double. The forEach method will then display these values:

```
stream
        .mapToDouble(e-> 2 * e)
        .forEach(e -> out.printf("%.4f  ", e));
```

The cascading of method invocations is referred to as **fluent programing**.

Using Java 8 to perform matrix multiplication

Here, we will illustrate how streams can be used to perform matrix multiplication. The definitions of the A, B, and C matrices are the same as declared in the *Implementing basic matrix operations* section. They are duplicated here for your convenience:

```
double A[][] = {
    {0.1950, 0.0311},
    {0.3588, 0.2203},
    {0.1716, 0.5931},
    {0.2105, 0.3242}};
double B[][] = {
    {0.0502, 0.9823, 0.9472},
    {0.5732, 0.2694, 0.916}};
double C[][] = new double[n][p];
```

The following sequence is a stream implementation of matrix multiplication. A detailed explanation of the code follows:

```
C = Arrays.stream(A)
        .parallel()
        .map(AMatrixRow -> IntStream.range(0, B[0].length)
                .mapToDouble(i -> IntStream.range(0, B.length)
                        .mapToDouble(j -> AMatrixRow[j] * B[j][i])
                        .sum()
                ).toArray()).toArray(double[][]::new);
```

The first `map` method, shown as follows, creates a stream of double vectors representing the 4 rows of the A matrix. The `range` method will return a list of stream elements ranging from its first argument to the second argument.

```
.map(AMatrixRow -> IntStream.range(0, B[0].length)
```

The variable `i` corresponds to the numbers generated by the second `range` method, which corresponds to the number of rows in the B matrix (2). The variable `j` corresponds to the numbers generated by the third `range` method, representing the number of columns of the B matrix (3).

At the heart of the statement is the matrix multiplication, where the `sum` method calculates the sum:

```
.mapToDouble(j -> AMatrixRow[j] * B[j][i])
.sum()
```

The last part of the expression creates the two-dimensional array for the C matrix. The operator, `::new`, is called a method reference and is a shorter way of invoking the new operator to create a new object:

```
).toArray()).toArray(double[][]::new);
```

The `displayResult` method is as follows:

```
public void displayResult() {
    out.println("Result");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            out.printf("%.4f  ", C[i][j]);
        }
        out.println();
    }
}
```

The output of this sequence follows:

```
Result
0.0276  0.1999  0.2132
0.1443  0.4118  0.5417
0.3486  0.3283  0.7058
0.1964  0.2941  0.4964
```

Using Java 8 to perform map-reduce

In the next section, we will use Java 8 streams to perform a map-reduce operation similar to the one demonstrated using Hadoop in the *Using map-reduce* section. In this example, we will use a `Stream` of `Book` objects. We will then demonstrate how to use the Java 8 `reduce` and `average` methods to get our total page count and average page count.

Rather than begin with a text file, as we did in the Hadoop example, we have created a `Book` class with title, author, and page-count fields. In the `main` method of the `driver` class, we have created new instances of `Book` and added them to an `ArrayList` called `books`. We have also created a `double` value `average` to hold our average, and initialized our variable `totalPg` to zero:

```
ArrayList<Book> books = new ArrayList<>();
double average;
int totalPg = 0;

books.add(new Book("Moby Dick", "Herman Melville", 822));
books.add(new Book("Charlotte's Web", "E.B. White", 189));
books.add(new Book("The Grapes of Wrath", "John Steinbeck", 212));
books.add(new Book("Jane Eyre", "Charlotte Bronte", 299));
books.add(new Book("A Tale of Two Cities", "Charles Dickens", 673));
books.add(new Book("War and Peace", "Leo Tolstoy", 1032));
books.add(new Book("The Great Gatsby", "F. Scott Fitzgerald",    275));
```

Next, we perform a map and reduce operation to calculate the total number of pages in our set of books. To accomplish this in a parallel manner, we use the `stream` and `parallel` methods. We then use the `map` method with a lambda expression to accumulate all of the page counts from each `Book` object. Finally, we use the `reduce` method to merge our page counts into one final value, which is to be assigned to `totalPg`:

```
totalPg = books
        .stream()
        .parallel()
        .map((b) -> b.pgCnt)
        .reduce(totalPg, (accumulator, _item) -> {
            out.println(accumulator + " " +_item);
            return accumulator + _item;
                });
```

Notice in the preceding `reduce` method we have chosen to print out information about the reduction operation's cumulative value and individual items. The `accumulator` represents the aggregation of our page counts. The `_item` represents the individual task within the map-reduce process undergoing reduction at any given moment.

In the output that follows, we will first see the `accumulator` value stay at zero as each individual book item is processed. Gradually, the `accumulator` value increases. The final operation is the reduction of the values `1223` and `2279`. The sum of these two numbers is `3502`, or the total page count for all of our books:

```
0 822
0 189
0 299
0 673
0 212
299 673
0 1032

0 275
1032 275
972 1307
189 212
822 401
1223 2279
```

Next, we will add code to calculate the average page count of our set of books. We multiply our `totalPg` value, determined using map-reduce, by `1.0` to prevent truncation when we divide by the integer returned by the `size` method. We then print out `average`.

```
average = 1.0 * totalPg / books.size();
out.printf("Average Page Count: %.4f\n", average);
```

Our output is as follows:

Average Page Count: 500.2857

We could have used Java 8 streams to calculate the average directly using the `map` method. Add the following code to the `main` method. We use `parallelStream` with our `map` method to simultaneously get the page count for each of our books. We then use `mapToDouble` to ensure our data is of the correct type to calculate our average. Finally, we use the `average` and `getAsDouble` methods to calculate our average page count:

```
average = books
        .parallelStream()
        .map(b -> b.pgCnt)
        .mapToDouble(s -> s)
        .average()
        .getAsDouble();
out.printf("Average Page Count: %.4f\n", average);
```

Then we print out our average. Our output, identical to our previous example, is as follows:

Average Page Count: 500.2857

These techniques made use of Java 8 capabilities related to the map-reduce framework to solve numeric problems. This type of process can also be applied to other types of data, including text-based data. The true benefit is seen when these processes handle extremely large datasets within a greatly reduced time frame.

Summary

Data science uses math extensively to analyze problems. There are numerous Java math libraries available, many of which support concurrent operations. In this chapter, we introduced a number of libraries and techniques to provide some insight into how they can be used to support and improve the performance of applications.

We started with a discussion of how simple matrix multiplication is performed. A basic Java implementation was presented. In later sections, we duplicated the implementation using other APIs and technologies.

Many higher level APIs, such as DL4J, support a number of useful data analysis techniques. Beneath these APIs often lies concurrent support for multiple CPUs and GPUs. Sometimes this support is configurable, as is the case for DL4J. We briefly discussed how we can configure ND4J to support multiple processors.

The map-reduce algorithm has found extensive use in the data science community. We took advantage of the parallel processing power of this framework to calculate the average of a given set of values, the page counts for a set of books. This technique used Apache's Hadoop to perform the map and reduce functions.

Mathematical techniques are supported by a large number of libraries. Many of these libraries do not directly support parallel operations. However, understanding what is available and how they can be used is important. To that end, we demonstrated how three different Java APIs can be used: jblas, Apache Commons Math, and ND4J.

OpenCL is an API that supports parallel operations on a variety of hardware platforms, processor types, and languages. This support is fairly low level. There are a number of Java bindings for OpenCL, which we reviewed.

Aparapi is a higher level of support for Java that can use CPUs, CUDA, or OpenCL to effect parallel operations. We demonstrated this support using the matrix multiplication example.

We wrapped up our discussion with an introduction to Java 8 streams and lambda expressions. These language elements can support parallel operations to improve an application's performance. In addition, this can often provide a more elegant and more maintainable implementation once the programmer becomes familiar with the techniques. We also demonstrated techniques for performing map-reduce using Java 8.

In the next chapter, we will conclude the book by illustrating how many of the techniques introduced can be used to build a complete application.

Chapter 12. Bringing It All Together

While we have demonstrated many aspects of using Java to support data science tasks, the need to combine and use these techniques in an integrated manner exists. It is one thing to use the techniques in isolation and another to use them in a cohesive fashion. In this chapter, we will provide you with additional experience with these technologies and insights into how they can be used together.

Specifically, we will create a console-based application that analyzes tweets related to a user-defined topic. Using a console-based application allows us to focus on data-science-specific technologies and avoids having to choose a specific GUI technology that may not be relevant to us. It provides a common base from which a GUI implementation can be created if needed.

The application performs and illustrates the following high-level tasks:

- Data acquisition
- Data cleaning, including:
  - Removing stop words
  - Cleaning the text
    - Sentiment analysis
    - Basic data statistic collection
    - Display of results

More than one type of analysis can be used with many of these steps. We will show the more relevant approaches and allude to other possibilities as appropriate. We will use Java 8's features whenever possible.

Defining the purpose and scope of our application

The application will prompt the user for a set of selection criteria, which include topic and sub-topic areas, and the number of tweets to process. The analysis performed will simply compute and display the number of positive and negative tweets for a topic and sub-topic. We used a generic sentiment analysis model, which will affect the quality of the sentiment analysis. However, other models and more analysis can be added.

We will use a Java 8 stream to structure the processing of tweet data. It is a stream of `TweetHandler` objects, as we will describe shortly.

We use several classes in this application. They are summarized here:

- `TweetHandler`: This class holds the raw tweet text and specific fields needed for the processing including the actual tweet, username, and similar attributes.
- `TwitterStream`: This is used to acquire the application's data. Using a specific class separates the acquisition of the data from its processing. The class possesses a few fields that control how the data is acquired.
- `ApplicationDriver`: This contains the `main` method, user prompts, and the `TweetHandler` stream that controls the analysis.

Each of these classes will be detailed in later sections. However, we will present `ApplicationDriver` next to provide an overview of the analysis process and how the user interacts with the application.

Understanding the application's architecture

Every application has its own unique structure, or architecture. This architecture provides the overarching organization or framework for the application. For this application, we combine the three classes using a Java 8 stream in the `ApplicationDriver` class. This class consists of three methods:

- `ApplicationDriver`: Contains the applications' user input
- `performAnalysis`: Performs the analysis
- `main`: Creates the `ApplicationDriver` instance

The class structure is shown next. The three instance variables are used to control the processing:

```java
public class ApplicationDriver {
    private String topic;
    private String subTopic;
    private int numberOfTweets;

    public ApplicationDriver() { ... }
    public void performAnalysis() { ...       }

    public static void main(String[] args) {
        new ApplicationDriver();
    }
}
```

The `ApplicationDriver` constructor follows. A `Scanner` instance is created and the sentiment analysis model is built:

```java
public ApplicationDriver() {
    Scanner scanner = new Scanner(System.in);
    TweetHandler swt = new TweetHandler();
    swt.buildSentimentAnalysisModel();
    ...
}
```

The remainder of the method prompts the user for input and then calls the `performAnalysis` method:

```java
out.println("Welcome to the Tweet Analysis Application");
out.print("Enter a topic: ");
this.topic = scanner.nextLine();
out.print("Enter a sub-topic: ");
this.subTopic = scanner.nextLine().toLowerCase();
out.print("Enter number of tweets: ");
this.numberOfTweets = scanner.nextInt();
performAnalysis();
```

The `performAnalysis` method uses a Java 8 `Stream` instance obtained from the `TwitterStream` instance. The `TwitterStream` class constructor uses the number of tweets and `topic` as input. This class is discussed in the *Data acquisition using Twitter* section:

```
public void performAnalysis() {
Stream<TweetHandler> stream = new TwitterStream(
    this.numberOfTweets, this.topic).stream();
    ...
}
```

The stream uses a series of `map`, `filter`, and a `forEach` method to perform the processing. The `map` method modifies the stream's elements. The `filter` methods remove elements from the stream. The `forEach` method will terminate the stream and generate the output.

The individual methods of the stream are executed in order. When acquired from a public Twitter stream, the Twitter information arrives as a JSON document, which we process first. This allows us to extract relevant tweet information and set the data to fields of the `TweetHandler` instance. Next, the text of the tweet is converted to lowercase. Only English tweets are processed and only those tweets that contain the sub-topic will be processed. The tweet is then processed. The last step computes the statistics:

```
stream
        .map(s -> s.processJSON())
        .map(s -> s.toLowerCase())
        .filter(s -> s.isEnglish())
        .map(s -> s.removeStopWords())
        .filter(s -> s.containsCharacter(this.subTopic))
        .map(s -> s.performSentimentAnalysis())
        .forEach((TweetHandler s) -> {
            s.computeStats();
            out.println(s);
        });
```

The results of the processing are then displayed:

```
out.println();
out.println("Positive Reviews: "
        + TweetHandler.getNumberOfPositiveReviews());
out.println("Negative Reviews: "
        + TweetHandler.getNumberOfNegativeReviews());
```

We tested our application on a Monday night during a Monday-night football game and used the topic #MNF. The # symbol is called a **hashtag** and is used to categorize tweets. By selecting a popular category of tweets, we ensured that we would have plenty of Twitter data to work with. For simplicity, we chose the football subtopic. We also chose to only analyze 50 tweets for this example. The following is an abbreviated sample of our prompts, input, and output:

```
Building Sentiment Model
Welcome to the Tweet Analysis Application
Enter a topic: #MNF
Enter a sub-topic: football
Enter number of tweets: 50
Creating Twitter Stream
51 messages processed!
Text: rt @ bleacherreport : touchdown , broncos ! c . j . anderson punches !
lead , 7 - 6 # mnf # denvshou
```

```
Date: Mon Oct 24 20:28:20 CDT 2016
Category: neg
...
Text: i cannot emphasize enough how big td drive . @ broncos offense . needed
confidence booster & amp ; just got . # mnf # denvshou
Date: Mon Oct 24 20:28:52 CDT 2016
Category: pos
Text: least touchdown game . # mnf
Date: Mon Oct 24 20:28:52 CDT 2016
Category: neg
Positive Reviews: 13
Negative Reviews: 27
```

We print out the text of each tweet, along with a timestamp and category. Notice that the text of the tweet does not always make sense. This may be due to the abbreviated nature of Twitter data, but it is partially due to the fact this text has been cleaned and stop words have been removed. We should still see our topic, #MNF, although it will be lowercase due to our text cleaning. At the end, we print out the total number of tweets classified as positive and negative.

The classification of tweets is done by the performSentimentAnalysis method. Notice the process of classification using sentiment analysis is not always precise. The following tweet mentions a touchdown by a Denver Broncos player. This tweet could be construed as positive or negative depending on an individual's personal feelings about that team, but our model classified it as positive:

```
Text: cj anderson td run @ broncos . broncos now lead 7 - 6 . # mnf
Date: Mon Oct 24 20:28:42 CDT 2016
Category: pos
```

Additionally, some tweets may have a neutral tone, such as the one shown next, but still be classified as either positive or negative. The following tweet is a retweet of a popular sports news twitter handle, @bleacherreport:

```
Text: rt @ bleacherreport : touchdown , broncos ! c . j . anderson punches !
lead , 7 - 6 # mnf # denvshou
Date: Mon Oct 24 20:28:37 CDT 2016
Category: neg
```

This tweet has been classified as negative but perhaps could be considered neutral. The contents of the tweet simply provide information about a score in a football game. Whether this is a positive or negative event will depend upon which team a person may be rooting for. When we examine the entire set of tweet data analysed, we notice that this same @bleacherreport tweet has been retweeted a number of times and classified as negative each time. This could skew our analysis when we consider that we may have a large number of improperly classified tweets. Using incorrect data decreases the accuracy of the results.

One option, depending on the purpose of analysis, may be to exclude tweets by news outlets or other popular Twitter users. Additionally we could exclude tweets with *RT*, an abbreviation denoting that the tweet is a retweet of another user.

There are additional issues to consider when performing this type of analysis, including the sub-topic used. If we were to analyze the popularity of a Star Wars character, then we would need to be careful which names we use. For example, when choosing a character name such as Han Solo, the tweet may use an alias. Aliases for Han Solo include Vykk Draygo, Rysto, Jenos Idanian, Solo Jaxal, Master Marksman, and Jobekk Jonn, to mention a few (http://starwars.wikia.com/wiki/Category:Han_Solo_aliases). The actor's name may be used instead of the actual character, which is Harrison Ford in the case of Han Solo. We may also want to consider the actor's nickname, such as Harry for Harrison.

Data acquisition using Twitter

The Twitter API is used in conjunction with HBC's HTTP client to acquire tweets, as previously illustrated in the Handling Twitter section of [Chapter 2](#), *Data Acquisition*. This process involves using the public stream API at the default access level to pull a sample of public tweets currently streaming on Twitter. We will refine the data based on user-selected keywords.

To begin, we declare the `TwitterStream` class. It consists of two instance variables, (`numberOfTweets` and `topic`), two constructors, and a `stream` method. The `numberOfTweets` variable contains the number of tweets to select and process, and `topic` allows the user to search for tweets related to a specific topic. We have set our default constructor to pull `100` tweets related to `Star Wars`:

```
public class TwitterStream {
    private int numberOfTweets;
    private String topic;

    public TwitterStream() {
        this(100, "Stars Wars");
    }

    public TwitterStream(int numberOfTweets, String topic) { ... }

}
```

The heart of our `TwitterStream` class is the `stream` method. We start by performing authentication using the information provided by Twitter when we created our Twitter application. We then create a `BlockingQueue` object to hold our streaming data. In this example, we will set a default capacity of `1000`. We use our `topic` variable in the `trackTerms` method to specify the types of tweets we are searching for. Finally, we specify our `endpoint` and turn off stall warnings:

```
String myKey = "mySecretKey";
String mySecret = "mySecret";
String myToken = "myToKen";
String myAccess = "myAccess";

out.println("Creating Twitter Stream");
BlockingQueue<String> statusQueue = new
LinkedBlockingQueue<>(1000);
StatusesFilterEndpoint endpoint = new StatusesFilterEndpoint();
endpoint.trackTerms(Lists.newArrayList("twitterapi", this.topic));
endpoint.stallWarnings(false);
```

Now we can create an `Authentication` object using `OAuth1`, a variation of the `OAuth` class. This allows us to build our connection client and complete the HTTP connection:

```
Authentication twitterAuth = new OAuth1(myKey, mySecret, myToken,
  myAccess);
```

```
BasicClient twitterClient = new ClientBuilder()
        .name("Twitter client")
        .hosts(Constants.STREAM_HOST)
        .endpoint(endpoint)
        .authentication(twitterAuth)
        .processor(new StringDelimitedProcessor(statusQueue))
        .build();

twitterClient.connect();
```

Next, we create two ArrayLists, `list` to hold our `TweetHandler` objects and `twitterList` to hold the JSON data streamed from Twitter. We will discuss the `TweetHandler` object in the next section. We use the `drainTo` method in place of the `poll` method demonstrated in , *Data Acquisition*, because it can be more efficient for large amounts of data:

```
List<TweetHandler> list = new ArrayList();
List<String> twitterList = new ArrayList();
```

Next we loop through our retrieved messages. We call the `take` method to remove each string message from the `BlockingQueue` instance. We then create a new `TweetHandler` object using the message and place it in our `list`. After we have handled all of our messages and the for loop completes, we stop the HTTP client, display the number of messages, and return our stream of `TweetHandler` objects:

```
statusQueue.drainTo(twitterList);
for(int i=0; i<numberOfTweets; i++) {
    String message;
    try {
        message = statusQueue.take();
        list.add(new TweetHandler(message));
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
twitterClient.stop();
out.printf("%d messages processed!\n",
    twitterClient.getStatsTracker().getNumMessages());

return list.stream();
}
```

We are now ready to clean and analyze our data.

Understanding the TweetHandler class

The `TweetHandler` class holds information about a specific tweet. It takes the raw JSON tweet and extracts those parts that are relevant to the application's needs. It also possesses the methods to process the tweet's text such as converting the text to lowercase and removing tweets that are not relevant. The first part of the class is shown next:

```
public class TweetHandler {
    private String jsonText;
    private String text;
    private Date date;
    private String language;
    private String category;
    private String userName;
    ...
    public TweetHandler processJSON() { ... }
    public TweetHandler toLowerCase(){ ... }
    public TweetHandler removeStopWords(){ ... }
    public boolean isEnglish(){ ... }
    public boolean containsCharacter(String character) { ... }
    public void computeStats(){ ... }
    public void buildSentimentAnalysisModel{ ... }
    public TweetHandler performSentimentAnalysis(){ ... }
}
```

The instance variables show the type of data retrieved from a tweet and processed, as detailed here:

- `jsonText`: The raw JSON text
- `text`: The text of the processed tweet
- `date`: The date of the tweet
- `language`: The language of the tweet
- `category`: The tweet classification, which is positive or negative
- `userName`: The name of the Twitter user

There are several other instance variables used by the class. The following are used to create and use a sentiment analysis model. The classifier static variable refers to the model:

```
private static String[] labels = {"neg", "pos"};
private static int nGramSize = 8;
private static DynamicLMClassifier<NGramProcessLM>
    classifier = DynamicLMClassifier.createNGramProcess(
        labels, nGramSize);
```

The default constructor is used to provide an instance to build the sentiment model. The single argument constructor creates a `TweetHandler` object using the raw JSON text:

```
public TweetHandler() {
    this.jsonText = "";
}
```

```
public TweetHandler(String jsonText) {
    this.jsonText = jsonText;
}
```

The remainder of the methods are discussed in the following sections.

Extracting data for a sentiment analysis model

In [Chapter 9](#), *Text Analysis*, we used DL4J to perform sentiment analysis. We will use LingPipe in this example as an alternative to our previous approach. Because we want to classify Twitter data, we chose a dataset with pre-classified tweets, available at http://thinknook.com/wp-content/uploads/2012/09/Sentiment-Analysis-Dataset.zip. We must complete a one-time process of extracting this data into a format we can use with our model before we continue with our application development.

This dataset exists in a large `.csv` file with one tweet and classification per line. The tweets are classified as either `0` (negative) or `1` (positive). The following is an example of one line of this data file:

```
95,0,Sentiment140, - Longest night ever.. ugh!    http://tumblr.com/xwp1yxhi6
```

The first element represents a unique ID number which is part of the original data set and which we will use for the filename. The second element is the classification, the third is a data set label (effectively ignored for the purposes of this project), and the last element is the actual tweet text. Before we can use this data with our LingPipe model, we must write each tweet into an individual file. To do this, we created three string variables. The `filename` variable will be assigned either `pos` or `neg` depending on each tweet's classification and will be used in the write operation. We also use the `file` variable to hold the name of the individual tweet file and the `text` variable to hold the individual tweet text. Next, we use the `readAllLines` method with the `Paths` class's `get` method to store our data in a `List` object. We need to specify the charset, `StandardCharsets.ISO_8859_1`, as well:

```
try {
    String filename;
    String file;
    String text;
    List<String> lines = Files.readAllLines(
Paths.get("\\path-to-file\\SentimentAnalysisDataset.csv"),
StandardCharsets.ISO_8859_1);
    ...

} catch (IOException ex) {
    // Handle exceptions
}
```

Now we can loop through our list and use the `split` method to store our `.csv` data in a string array. We convert the element at position `1` to an integer and determine whether it is a `1`. Tweets classified with a `1` are considered positive tweets and we set `filename` to `pos`. All other tweets set the `filename` to `neg`. We extract the output filename from the element at position `0` and the text from element `3`. We ignore the label in position `2` for the purposes of this project. Finally, we write out our data:

```
for(String s : lines) {
    String[] oneLine = s.split(",");
    if(Integer.parseInt(oneLine[1])==1) {
```

```
        filename = "pos";
    } else {
        filename = "neg";
    }
    file = oneLine[0]+".txt";
    text = oneLine[3];
    Files.write(Paths.get(
        path-to-file\\txt_sentoken"+filename+""+file),
        text.getBytes());
}
```

Notice that we created the neg and pos directories within the txt_sentoken directory. This location is important when we read the files to build our model.

Building the sentiment model

Now we are ready to build our model. We loop through the `labels` array, which contains `pos` and `neg`, and for each label we create a new `Classification` object. We then create a new file using this label and use the `listFiles` method to create an array of filenames. Next, we will traverse these filenames using a `for` loop:

```
public void buildSentimentAnalysisModel() {
    out.println("Building Sentiment Model");

    File trainingDir = new File("\\path to file\\txt_sentoken");
    for (int i = 0; i < labels.length; i++) {
        Classification classification =
            new Classification(labels[i]);
        File file = new File(trainingDir, labels[i]);
        File[] trainingFiles = file.listFiles();
        ...
    }
}
```

Within the `for` loop, we extract the tweet data and store it in our string, `review`. We then create a new `Classified` object using `review` and `classification`. Finally we can call the `handle` method to classify this particular text:

```
for (int j = 0; j < trainingFiles.length; j++) {
    try {
        String review = Files.readFromFile(trainingFiles[j],
            "ISO-8859-1");
        Classified<CharSequence> classified = new
            Classified<>(review, classification);
        classifier.handle(classified);
    } catch (IOException ex) {
        // Handle exceptions
    }
 }
```

For the dataset discussed in the previous section, this process may take a substantial amount of time. However, we consider this time trade-off to be worth the quality of analysis made possible by this training data.

Processing the JSON input

The Twitter data is retrieved using JSON format. We will use Twitter4J (http://twitter4j.org) to extract the relevant parts of the tweet and store in the corresponding field of the `TweetHandler` class.

The `TweetHandler` class's `processJSON` method does the actual data extraction. An instance of the `JSONObject` is created based on the JSON text. The class possesses several methods to extract specific types of data from an object. We use the `getString` method to get the fields we need.

The start of the `processJSON` method is shown next, where we start by obtaining the `JSONObject` instance, which we will use to extract the relevant parts of the tweet:

```
public TweetHandler processJSON() {
    try {
        JSONObject jsonObject = new JSONObject(this.jsonText);
        ...
    } catch (JSONException ex) {
        // Handle exceptions
    }
    return this;
}
```

First, we extract the tweet's text as shown here:

```
this.text = jsonObject.getString("text");
```

Next, we extract the tweet's date. We use the `SimpleDateFormat` class to convert the date string to a `Date` object. Its constructor is passed a string that specifies the format of the date string. We used the string `"EEE MMM d HH:mm:ss Z yyyy"`, whose parts are detailed next. The order of the string elements corresponds to the order found in the JSON entity:

- `EEE`: Day of the week specified using three characters
- `MMM`: Month, using three characters
- `d`: Day of the month
- `HH:mm:ss`: Hours, minutes, and seconds
- `Z`: Time zone
- `yyyy`: Year

The code follows:

```
SimpleDateFormat sdf = new SimpleDateFormat(
    "EEE MMM d HH:mm:ss Z yyyy");
try {
    this.date = sdf.parse(jsonObject.getString("created_at"));
} catch (ParseException ex) {
    // Handle exceptions
}
```

The remaining fields are extracted as shown next. We had to extract an intermediate JSON object to extract the `name` field:

```
this.language = jsonObject.getString("lang");
JSONObject user = jsonObject.getJSONObject("user");
this.userName = user.getString("name");
```

Having acquired and extracted the text, we are now ready to perform the important task of cleaning the data.

Cleaning data to improve our results

Data cleaning is a critical step in most data science problems. Data that is not properly cleaned may have errors such as misspellings, inconsistent representation of elements such as dates, and extraneous words.

There are numerous data cleaning options that we can apply to Twitter data. For this application, we perform simple cleaning. In addition, we will filter out certain tweets.

The conversion of the text to lowercase letters is easily achieved as shown here:

```java
public TweetHandler toLowerCase() {
    this.text = this.text.toLowerCase().trim();
    return this;
}
```

Part of the process is to remove certain tweets that are not needed. For example, the following code illustrates how to detect whether the tweet is in English and whether it contains a sub-topic of interest to the user. The `boolean` return value is used by the `filter` method in the Java 8 stream, which performs the actual removal:

```java
public boolean isEnglish() {
    return this.language.equalsIgnoreCase("en");
}

public boolean containsCharacter(String character) {
    return this.text.contains(character);
}
```

Numerous other cleaning operations can be easily added to the process such as removing leading and trailing white space, replacing tabs, and validating dates and email addresses.

Removing stop words

Stop words are those words that do not contribute to the understanding or processing of data. Typical stop words include the 0, and, a, and or. When they do not contribute to the data process, they can be removed to simplify processing and make it more efficient.

There are several techniques for removing stop words, as discussed in Chapter 9, *Text Analysis*. For this application, we will use LingPipe (http://alias-i.com/lingpipe/) to remove stop words. We use the `EnglishStopTokenizerFactory` class to obtain a model for our stop words based on an `IndoEuropeanTokenizerFactory` instance:

```
public TweetHandler removeStopWords() {
    TokenizerFactory tokenizerFactory
            = IndoEuropeanTokenizerFactory.INSTANCE;
    tokenizerFactory =
        new EnglishStopTokenizerFactory(tokenizerFactory);
    ...
    return this;
}
```

A series of tokens that do not contain stop words are extracted, and a `StringBuilder` instance is used to create a string to replace the original text:

```
Tokenizer tokens = tokenizerFactory.tokenizer(
        this.text.toCharArray(), 0, this.text.length());
StringBuilder buffer = new StringBuilder();
for (String word : tokens) {
    buffer.append(word + " ");
}
this.text = buffer.toString();
```

The LingPipe model we used may not be the best suited for all tweets. In addition, it has been suggested that removing stop words from tweets may not be productive (http://oro.open.ac.uk/40666/). Options to select various stop words and whether stop words should even be removed can be added to the stream process.

Performing sentiment analysis

We can now perform sentiment analysis using the model built in the Building the sentiment model section of this chapter. We create a new `Classification` object by passing our cleaned text to the `classify` method. We then use the `bestCategory` method to classify our text as either positive or negative. Finally, we set `category` to the result and return the `TweetHandler` object:

```
public TweetHandler performSentimentAnalysis() {
    Classification classification =
        classifier.classify(this.text);
    String bestCategory = classification.bestCategory();
    this.category = bestCategory;
    return this;
}
```

We are now ready to analyze the results of our application.

Analysing the results

The analysis performed in this application is fairly simple. Once the tweets have been classified as either positive or negative, a total is computed. We used two static variables for this purpose:

```
    private static int numberOfPositiveReviews = 0;
    private static int numberOfNegativeReviews = 0;
```

The `computeStats` method is called from the Java 8 stream and increments the appropriate variable:

```java
public void computeStats() {
    if(this.category.equalsIgnoreCase("pos")) {
        numberOfPositiveReviews++;
    } else {
        numberOfNegativeReviews++;
    }
}
```

Two `static` methods provide access to the number of reviews:

```java
public static int getNumberOfPositiveReviews() {
    return numberOfPositiveReviews;
}

public static int getNumberOfNegativeReviews() {
    return numberOfNegativeReviews;
}
```

In addition, a simple `toString` method is provided to display basic tweet information:

```java
public String toString() {
    return "\nText: " + this.text
            + "\nDate: " + this.date
            + "\nCategory: " + this.category;
}
```

More sophisticated analysis can be added as required. The intent of this application was to demonstrate a technique for combining the various data processing tasks.

Other optional enhancements

There are numerous improvements that can be made to the application. Many of these are user preferences and others relate to improving the results of the application. A GUI interface would be useful in many situations. Among the user options, we may want add support for:

- Displaying individual tweets
- Allowing null sub-topics
- Processing other tweet fields
- Providing list of topics or sub-topics the user can choose from
- Generating additional statistics and supporting charts

With regard to process result improvements, the following should be considered:

- Correct user entries for misspelling
- Remove spacing around punctuation
- Use alternate stop word removal techniques
- Use alternate sentiment analysis techniques

The details of many of these enhancements are dependent on the GUI interface used and the purpose and scope of the application.

Summary

The intent of this chapter was to illustrate how various data science tasks can be integrated into an application. We chose an application that processes tweets because it is a popular social medium and allows us to apply many of the techniques discussed in earlier chapters.

A simple console-based interface was used to avoid cluttering the discussion with specific but possibly irrelevant GUI details. The application prompted the user for a Twitter topic, a sub-topic, and the number of tweets to process. The analysis consisted of determining the sentiments of the tweets, with simple statistics regarding the positive or negative nature of the tweets.

The first step in the process was to build a sentiment model. We used LingPipe classes to build a model and perform the analysis. A Java 8 stream was used and supported a fluent style of programming where the individual processing steps could be easily added and removed.

Once the stream was created, the JSON raw text was processed and used to initialize a `TweetHandler` class. Instances of this class were subsequently modified, including converting the text to lowercase, removing non-English tweets, removing stop words, and selecting only those tweets that contain the sub-topic. Sentiment analysis was then performed, followed by the computation of the statistics.

Data science is a broad topic that utilizes a wide range of statistical and computer science topics. In this book, we provided a brief introduction to many of these topics and how they are supported by Java.