Community Experience Distilled

Learning Haskell Data Analysis

Analyze, manipulate, and process datasets of varying sizes efficiently using Haskell

James Church

[PACKT] PUBLISHING open source* community experience distilled

Learning Haskell Data Analysis

Analyze, manipulate, and process datasets of varying sizes efficiently using Haskell

James Church

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Learning Haskell Data Analysis

First published: May 2015

Production reference: 1250515

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

ISBN 978-1-78439-470-7

www.packtpub.com

Credits

Author
James Church

Reviewers
Joseph Adams
William Kong
Samuli Thomasson

Commissioning Editor
Ashwin Nair

Acquisition Editor
Shaon Basu

Content Development Editor
Parita Khedekar

Technical Editor
Gaurav Suri

Copy Editors
Aditya Nair
Vedangi Narvekar

Project Coordinator
Milton Dsouza

Proofreaders
Stephen Copestake
Safis Editing

Indexer
Priya Sane

Graphics
Sheetal Aute
Jason Monteiro
Abhinash Sahu

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

About the Author

James Church is an assistant professor of computer science at the University of West Georgia. James completed his PhD in computer science from the University of Mississippi under the advisement of Dr. Yixin Chen, with a research focus on computational geometry.

While at the University of Mississippi, he learned the skills necessary for data analysis in his side job, where he worked as a database administrator and analyst for the Marijuana Potency Monitoring Program (MPMP) led by Dr. Mahmoud ElSohly. The software written by James is used by the laboratory to store and track the chemical composition of marijuana samples. This data is provided to the United States National Institute on Drug Abuse to report marijuana potency.

The knowledge gained through his experience as an analyst for the MPMP (as well as other companies) was turned into a data analysis course for undergraduates at the University of Mississippi. This course was taught using the languages of Python and R.

James enjoys spending time with his wife, Michelle Allen, teaching, and playing board games with his friends.

I would like to thank the reviewers for testing and reshaping the code in these pages into something readable. While I made every attempt to write the best possible book on this topic, I'm sure that mistakes will be found. All mistakes are my own.

I would like to thank Michelle Allen. Your support made this book possible (Michelle drew the sketch of the airplane found in *Chapter 8*, *Building a Recommendation Engine*).

I would like to thank Dr. Yixin Chen, Dr. Dawn Wilkins, Dr. Conrad Cunningham, and the faculty of computer and information science at the University of Mississippi.

Finally, thanks, mom and dad.

About the Reviewers

Joseph Adams is a keen programmer, writing in Go, Haskell, Python, Java, and a few more programming languages. His interests include the designing of programming languages, especially the functional and esoteric ones, as well as the application of unusual data structures. When not programming, he enjoys reading books ranging from topics such as philosophy to logic and advanced mathematics. Joseph also regularly attends international computer conferences and has even spoken on his Scheme to Go compiler at Free and Open Source Software Developers' European Meeting (FOSDEM).

He has a number of public repositories on GitHub that you should definitely check out by visiting `http://github.com/jcla1`. He also blogs quite regularly at `http://jcla1.com`.

William Kong studied at the University of Waterloo, Canada, and holds a bachelor of mathematics degree with a specialization in mathematical finance and a minor in statistics. He has an extensive experience in the Haskell and R programming languages and has worked as a SAS statistical programmer and modeler in the financial risk modeling industry.

William's concentrations have been mainly in the fields of optimization, statistics, and computational mathematics. His current interests lie in the practical application of novel programming paradigms and languages, such as the ever-functional Haskell and the latest newcomer, Julia.

My gratitude goes out to my parents for their support and understanding during my long hours at my computer, reviewing and testing code. I must also acknowledge instrumental help from *Learn You a Haskell for Great Good: A Beginner's Guide*, *No Starch Press* and *Real World Haskell*, *O'Reilly Media*, which were an invaluable source of knowledge throughout the editing process.

Samuli Thomasson is a Haskell enthusiast who has written software in Haskell for over 3 years, mostly as a hobby and partly for his studies and work. He has built multiple web applications and miscellaneous tools, many of which were written in Haskell. He is currently interested in functional programming, distributed systems, data science, mathematics, and philosophy. He is a student at the Department of Computer Science in the University of Helsinki, and he also works in the Finnish software industry. While pursuing his budding career, he is actively searching for the best topics to learn and specialize in.

He lives in Helsinki, Finland, with his friends. You can take a look at his website by visiting `http://funktionaali.com`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

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

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book serves as an introduction to data analysis methods and practices from a computational and mathematical standpoint. *Data* is the collection of information within a particular domain of knowledge. The language of data analysis is mathematics. For the purposes of computation, we will use Haskell, the free, general-purpose language. The objective of each chapter is to solve a problem related to a common task in the craft of data analysis. The goals for this book are two-fold. The first goal is to help the reader gain confidence in working with large datasets. The second goal is to help the reader understand the mathematical nature of data. We don't just recommend libraries and functions in this book. Sometimes, we ignore popular libraries and write functions from scratch in order to demonstrate their underlying processes. By the end of this book, you should be able to solve seven common problems related to data analysis (one problem per chapter after the first chapter). You will also be equipped with a mental flowchart of the craft, from understanding and cleaning your dataset to asking testable questions about your dataset. We will stick to real-world problems and solutions. This book is your guide to your data.

What this book covers

Chapter 1, *Tools of the Trade*, discusses the software and the essential libraries used in the book. We will also solve two simple problems—how to find the median of a list of numbers and how to locate the vowels in a word. These problems serve as an introduction to working with small datasets. We also suggest two nonessential tools to assist you with the projects in this text—Git and Tmux.

Chapter 2, *Getting Our Feet Wet*, introduces you to csv files and SQLite3. CSV files are human- and machine-readable and are found throughout the Internet as a common format to share data. Unfortunately, they are difficult to work with in Haskell. We will introduce a module to convert csv files into SQLite3 databases, which are comparatively much easier to work with. We will obtain a small csv file from the US Geological Survey, convert this dataset to an SQLite3 database, and perform some analysis on the earthquake data.

Chapter 3, *Cleaning Our Datasets*, discusses the oh-so-boring, yet oh-so-necessary topic of data cleaning. We shouldn't take clean, polished datasets for granted. Time and energy must be spent on creating a metadata document for a dataset. An equal amount of time must also be spent cleaning this document. This involves looking for blank entries or entries that do not fit the standard that we defined in our metadata document. Most of the work in this area is performed with the help of regular expressions. Regular expressions are a powerful tool by which we can search and manipulate data.

Chapter 4, *Plotting*, looks at the plotting of data. It's often easier to comprehend a dataset visually than through raw numbers. Here, we will download the history of the publicly traded companies on the New York Stock Exchange and discuss the investment strategy of growth investing. To do this, we will visually compare the yearly growth rate of Google, Microsoft, and Apple. These three companies belong to a similar industry (technology) but have different growth rates. We will discuss the normalization function, which allows us to compare companies with different share prices on the same graph.

Chapter 5, *Hypothesis Testing*, trains us to be skeptical of our own claims so that we don't fall for the trap of fooling ourselves. We will give ourselves the challenge of detecting an unfair coin. Successive coin flips follow a particular pattern called the binomial distribution. We will discuss the mathematics behind detecting whether a particular coin is following this distribution or not. We will follow this up with a question about baseball—"Is there a benefit if one has home field advantage?" To answer this question, we will download baseball data and put this hypothesis to the test.

Chapter 6, *Correlation and Regression Analysis*, discusses regression analysis. Regression analysis is a tool by which we can interpolate data where there is none. In keeping with the baseball theme, we will try to measure how much benefit there is to scoring baseball runs and winning baseball games. We will compute the runs-per-game and the win percentage of every team in Major League Baseball for the 2014 season and evaluate who is overperforming and underperforming on the field. This technique is simple enough to be used on other sports teams for similar analysis.

Chapter 7, *Naive Bayes Classification of Twitter Data*, analyzes the tweets from the popular social networking site, Twitter. Twitter has broad international appeal and people from around the world use the site. Twitter's API allows us to look at the language of each tweet. Using the individual words and the identified language, we will build a Naive Bayes classifier to detect the language of the sentences based on a database of downloaded tweets.

Chapter 8, *Building a Recommendation Engine*, continues with the analysis of the Twitter data and helps us create our own recommendation engine. This recommendation will help users find other users with similar interests based on the frequency of the words used in their tweets. There is a lot of data in word frequencies and we don't need all of it. So, we will discuss a technique to reduce the dimensionality of our data called Principal Component Analysis (PCA). PCA engines are used to recommend similar products for you to purchase or watch movies on commercial websites. We will cover the math and the implementation of a recommendation engine from scratch.

In each chapter we will introduce new functions. These functions will be added to a module file titled *LearningDataAnalysis0X* (where X is the current chapter number). We will frequently use functions from the earlier chapters to solve the problem from the chapter at hand. It will help you follow the chapters of this book in order so that you know when special functions mentioned in this book have been introduced.

Appendix, *Regular Expressions in Haskell*, focuses on the use of regular expressions in Haskell. If you aren't familiar with regular expressions, this will be a short reference guide to their usage.

What you need for this book

The software required for this book is the Haskell platform, the cabal tool to install libraries (which comes with Haskell), as well as tools such as SQLite3, gnuplot, and the LAPACK library for linear algebra. The installation instructions for each piece of software are mentioned at the time when the software is needed.

We tried to be *cross-platform* in this book because Haskell is a cross-platform language. SQLite3 and gnuplot are available for the Windows, Mac, and Linux operating systems. One problem that we encountered while writing this book was the difficulty in installing LAPACK for Windows, which is used in *Chapter 8*, *Building a Recommendation Engine*. At the time of writing this book, it is possible to get LAPACK to run on Windows, but the instructions are not that clear and hence it is not recommended. Instead, we recommend Windows users install Debian or Ubuntu Linux using VM software (such as Oracle VirtualBox).

Who this book is for

If you are a developer, an analyst, or a data scientist who wants to learn data analysis methods using Haskell and its libraries, then this book is for you. Prior experience with Haskell and basic knowledge of data science will be beneficial.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `percentChange` function only computes a single percent change at a given point in our data."

A block of code is set as follows:

```
module LearningDataAnalysis04 where
import Data.List
import Database.HDBC.Sqlite3
import Database.HDBC
import Graphics.EasyPlot
import LearningDataAnalysis02
```

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

```
sudo apt-get install gnuplot
```

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: "On the **Historical Prices** page, identify the link that says **Download to Spreadsheet**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/4707OS_ColoredImages.pdf`.

Errata

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

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

Piracy

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

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

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

Questions

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

1
Tools of the Trade

Data analysis is the craft of sifting through data for the purpose of learning or decision making. To ease the difficulties of sifting through data, we rely on databases and our knowledge of programming. For nut-and-bolts coding, this text uses Haskell. For storage, plotting, and computations on large datasets, we will use SQLite3, gnuplot, and LAPACK respectively. These four pieces of software are a powerful combination that allow us to solve some difficult problems. In this chapter, we will discuss these tools of the trade and recommend a few more.

In this chapter, we will cover the following:

- Why we should consider Haskell for our next data analysis project
- Installing and configuring Haskell, the **GHCi** (short for **Glasgow Haskell Compiler interactive)** environment, and cabal
- The software packages needed in addition to Haskell: SQLite3, gnuplot, and LAPACK
- The *nearly* essential software packages that you should consider: Git and Tmux
- Our first program: computing the median of a list of values
- An introduction to the command-line environment

Welcome to Haskell and data analysis!

This book is about solving problems related to data. In each chapter, we will present a problem or a question that needs answering. The only way to get this answer is through an understanding of the data. Data analysis is not only a practice that helps us glean insight from information, but also an academic pursuit that combines knowledge of the basics of computer programming, statistics, machine learning, and linear algebra. The theory behind data analysis comes from statistics.

The concepts of summary statistics, sampling, and empirical testing are gifts from the statistical community. Computer science is a craft that helps us convert statistical procedures into formal algorithms, which are interpreted by a computer. Rarely will our questions about data be an end in themselves. Once the data has been analyzed, the analysis should serve as a plan to better decision-making powers. The field of machine learning is an attempt to create algorithms that are capable of making their own decisions based on the results of the analysis of a dataset. Finally, we will sometimes need to use linear algebra for complicated datasets. Linear algebra is the study of vector spaces and matrices, which can be understood by the data analyst as a multidimensional dataset with rows and columns. However, the most important skill of data analysts is their ability to communicate their findings with the help of a combination of written descriptions and graphs. Data science is a challenging field that requires a blend of computer science, mathematics, and statistics disciplines.

In the first chapter, the real-world problem is with regard to getting our environment ready. Many languages are suitable for data analysis, but this book tackles data problems using Haskell and assumes that you have a background in the Haskell language from *Chapter 2*, *Getting Our Feet Wet* onwards. If not, we encourage you to pick up a book on Haskell development. You can refer to *Learn You a Haskell for Great Good: A Beginner's Guide*, *Miran Lipovaca*, *No Starch Press*, and *Real World Haskell*, *Bryan O'Sullivan*, *John Goerzen*, *Donald Bruce Stewart*, *O'Reilly Media*, which are excellent texts if you want to learn programming in Haskell. *Learn You a Haskell for Great Good: A Beginner's Guide* can be read online at `http://learnyouahaskell.com/`. *Real World Haskell* can also be read online at `http://book.realworldhaskell.org/`. The former book is an introduction to the language, while the latter is a text on professional Haskell programming. Once you wade through these books (as well as *Learning Haskell Data Analysis*), we encourage you to read the book *Haskell Data Analysis Cookbook*, *Nishant Shukla*, *Packt Publishing*. This cookbook will provide snippets of code to work with a wide variety of data formats, databases, visualization tools, data structures, and clustering algorithms. We also recommend *Notes on Functional Programming with Haskell* by Dr. Conrad Cunningham.

Besides Haskell, we will discuss open source data formats, databases, and graphing software in the following manner:

- We will limit ourselves to working with two data serialization file formats: JSON and CSV. **CSV** is perhaps the most common data serialization format for uncompressed data with the weakness of not being an explicit standard. In a later chapter, we will examine data from the Twitter web service, which exports data in the **JSON** format. By limiting ourselves to two data formats, we will focus our efforts on problem solving instead of prolonged discussions of data formats.

- We will use SQLite3 for our database backend application. **SQLite3** is a lightweight database software that can store large amounts of data. Using a wrapper module, we can pull data directly from a SQLite3 database into the Haskell command line for analysis.
- We will use the EasyPlot Haskell wrapper module for **gnuplot**, which is a popular open source tool that is used to create publication-ready graphics. The **EasyPlot** wrapper provides access to a subset of features in gnuplot, but we shall see that this subset is more than sufficient for the creation of compelling graphs.

Why Haskell?

Since this is an introductory text to data analysis, we will focus on commonly used practices within the context of the Haskell language. Haskell is a general-purpose, purely functional programming language with strong static typing. Among the many features of Haskell are lazy evaluation, type inference, Lisp-like support for lists and tuples, and the Hackage repository. Here are the reasons why we like Haskell:

- Haskell has features that are similar to Lisp. These features are used to process lists of data (minus the syntax of Lisp). Higher-order functions such as `map`, `foldr`, `foldl`, and `filter` provide us with a standard interface to apply functions to lists. In Haskell, all the functions can be used as a parameter to the other functions, allowing the programmer to seamlessly manipulate data on the fly through anonymous functions known as lambda expressions. The `map` function is frequently used in this book due to the ease it provides in converting a list of elements of a data type to another type.
- Haskell is a purely functional programming language, and stateless in nature. This means that the only information known to a function is the information that is either passed into that function or returned from other function calls. The so-called variables are named after their mathematical properties and not the conventional computer programming sense of the word. Variables are not allowed to change. Instead, they are the bindings to expressions. Because of these limitations, functions are easier to test for their correctness than *stateful* languages.

- Haskell can handle large datasets that have a size that your system memory limitations will allow, which should be sufficient to handle most medium-sized data problems. Big data can be defined as any dataset that is so big that it needs to be broken up into pieces and aggregated in a secondary step or sampled prior to the analysis. A step-down of big data is medium data, which can be defined as a dataset that can be processed in its entirety without you having to break it into parts. There is no set number with regard to when a dataset grows in size from medium to big since ever-increasing hardware capabilities continuously redefine how much a computer can do. An informal definition of small data is a dataset that can be easily grasped in its entirety by a human, which can be considered to be a few numbers at best. All of the problems considered in this book were tested on a computer with a RAM of 2 GB. The smallest dataset examined in this chapter is 16 values and the largest dataset is about 7 MB in size. Each of the problems presented in this text should scale in size to the definition of medium data.
- Haskell enforces lazy evaluation. Lazy evaluation allows Haskell to delay the execution of a procedure until it is actually needed, for example, throughout this book, we will be setting up calculations over the course of several steps. In most *strict* languages, once these calculations are encountered by the language, they are immediately executed and the results are stored in memory. In lazy languages, commands are compiled and the system stores the instructions. If a calculation step is never used, it never gets evaluated, thus saving execution time. Once the calculation is required (for example, when we need to see the results displayed on the screen), only then will a lazy language evaluate the steps of our algorithm.
- Haskell supports type inference. Type inference allows Haskell to be strictly typed without having to declare the need for types as the code is being written, for example, consider the following `myFunc` function annotation:

  ```
  myFunc :: a -> a -> Integer
  ```

  This function requires two parameters, and it returns an `Integer`. The type is left ambiguous, and it will be inferred when the function is used. Because both the types are `a`, Haskell will use static type checking to ensure that the data type of the first parameter matches the data type of the second parameter. If we wish for the possibility of the first parameter to have a type that is different from the second, we can create a second inferred type named `b`. (Specific types begin with an uppercase letter and generic types must begin with a lowercase letter.)

- Using the cabal tool, a Haskell programmer has several thousands of libraries that can be quickly downloaded to a system. These libraries provide analysts with most of the common data analysis procedures. While many libraries exist within the cabal repository, sometimes we may opt not to use them in favor of an explicit description of the math and code behind a particular algorithm.

Getting ready

You will need to install the Haskell platform, which is available on all three major operating systems: Windows, Mac, and Linux. I primarily work with Debian Linux. Linux has the benefit of being equipped with a versatile command line, which can facilitate almost everything that is essential to the data analysis process. From the command line, we can download software, install Haskell libraries, download datasets, write files, and view raw datasets. An essential activity that the command line cannot do for us is the rendering of graphics that can be provided with sufficient detail to inspect rendered charts of our analyses.

Installing the Haskell platform on Linux

On Ubuntu- and Debian-based systems, you can install the Haskell platform using `apt-get`, as follows:

```
$ sudo apt-get install haskell-platform
```

This single command will install everything that is needed to get started, including the compiler (`ghc`), interactive command line (`ghci`), and the library install tool (`cabal`). Take a moment to test the following commands:

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.4.1
$ ghci --version
The Glorious Glasgow Haskell Compilation System, version 7.4.1
```

If you get back the version numbers for the Haskell compiler and the Haskell interactive prompt, you should be all set. However, we do need to perform some housekeeping with regards to cabal. We will use cabal throughout this book, and it will require an update immediately. We updated the cabal tool through `cabal` itself.

First, we will update the Haskell package list from Hackage using the `update` directive by using the following command:

```
$ cabal update
```

Next, we will download cabal using the `cabal-install` command. This command will not overwrite the existing cabal program. Instead, it will download an updated cabal to your `home` folder, which can be found at `~/.cabal/bin/cabal`.

```
$ cabal install cabal-install
```

Your system has two versions of cabal on it. We created an `alias` command to make sure that we only use the updated version of cabal. This is a temporary `alias` command. You should add the following line to one of your configuration files in your home directory. (We added ours to `~/.bash_aliases` and reloaded aliases with `source ~/.bash_aliases`.)

```
$ alias cabal='~/.cabal/bin/cabal'
```

If all goes according to plan, you will have an updated version of cabal on your system. Here is the version of cabal used at the time of writing this book:

```
$ cabal --version
cabal-install version 1.22.0.0
using version 1.22.0.0 of the Cabal library
```

If you use cabal long enough, you may run into problems. Rather than going into a prolonged discussion on how to manage Haskell packages, it is easier to start over with a clean slate. Your packages are downloaded to a folder under `~/.cabal`, and they are registered with the Haskell environment under the `~/.ghc/` directory. If you find that a package has not been installed due to a conflicted dependency, you can spend an evening reading the package documentation to figure out which packages need to be removed or installed. Alternatively, you can use the following command and wipe the slate clean:

```
$ rm -rf ~/.ghc
```

The preceding command wipes out all your installed Haskell packages. We can promise that you will not have conflicting packages if you have no packages. We call this the *Break Glass In Case of Emergency* solution. This is obviously is not the best solution, but it is a solution that gets your necessary packages installed. You have more important things to do than wrestle with cabal. While it may take about an hour or so to download and install packages with this approach, this approach is less stressful than the process of going through package version numbers.

The software used in addition to Haskell

There are three open source software packages used in this book that work alongside the Haskell platform. If you are using Debian or Ubuntu, you will be able to download each of these packages using the `apt-get` command-line tool. The instructions on how to download and install these packages will be introduced when the software is needed. If you are using Windows or Mac, you will have to consult the documentation for these software packages for an installation on your system.

SQLite3

SQLite3 (for more information refer to: `https://sqlite.org/`) is a standalone **Structured Query Language** (**SQL**) database engine. We use SQLite3 to filter and organize large amounts of data. It requires no configuration, does not use a background server process, and each database is self-contained in a single file ending with the `.sql` extension. The software is portable, has many features from the features found in sever-based SQL database engines, and can support large databases. We will introduce SQLite3 in *Chapter 2*, *Getting Our Feet Wet* and use it extensively in the rest of the book.

Gnuplot

Gnuplot (for more information refer to: `http://www.gnuplot.info/`) is a command-line tool that can be used to create charts and graphs for academic publications. It supports many features related to 2D and 3D plotting as well as a number of output and interactive formats. We will use gnuplot in conjunction with the EasyPlot Haskell wrapper module. EasyPlot gives us access to a subset of the features of gnuplot (which means that even though our charts are being piped through gnuplot, we will not be able to utilize the full power of gnuplot from within this library). Every chart presented in this book was created using EasyPlot and gnuplot.
We will introduce EasyPlot and gnuplot in *Chapter 4*, *Plotting*.

LAPACK

LAPACK (short for **Linear Algebra PACKage**) (for more information refer to: `http://www.netlib.org/lapack/`) has been constantly developed since the early 1990s. To this day, this library is written in FORTRAN. Since it is so vital to science, it is funded through the United States **National Science Foundation** (**NSF**). This library supports routines related to systems of equations such as matrix multiplication, matrix inversion, and eigenvalue decomposition. We will use the hmatrix wrapper for LAPACK in *Chapter 8*, *Building a Recommendation Engine* to write our own **Principal Component Analysis** (**PCA**) function to create a recommendation engine. We will also use LAPACK to avoid the messiness that comes when trying to write an eigensolver ourselves.

Nearly essential tools of the trade

This section is about the tools used in the preparation of this book. They aren't essential to Haskell or data analysis, but they deserve a mention.

Version control software – Git

If you have ever been in a situation where you needed to update an old file while keeping that old file, you may have been tempted to name the files `MyFileVersion1` and `MyFileVersion2`. In this instance, you used manual version control. Instead, you should use version control software.

Git is a distributed version control software that allows teams of programmers to work on a single project, track their changes, branch a project, merge project branches, and roll back mistakes if necessary. Git will scale from a team of 1 to hundreds of members.

If you already have a favorite software package for version control, we encourage you to use it while working through the examples in this book. If not, we will quickly demonstrate how to use Git.

First, you need to install Git by using the following code:

```
$ sudo apt-get install git
```

Git requires you to set up a repository in your working directory. Navigate to your folder for your Haskell project and create a repository:

```
$ git init
```

Once your repository is created, add the files that you've created in this chapter to the repository. Create a file called `LearningDataAnalysis01.hs`. At this point, the file should be blank. Let's add the blank file to our repository:

```
$ git add LearningDataAnalysis01.hs
```

Now, we'll commit the change:

```
$ git commit -m 'Add chapter 1 file'
```

Take a moment to revisit the `LearningDataAnalysis01.hs` file and make a change to damage the file. We can do this via the following command line:

```
$ echo "It was a mistake to add this line." >> LearningDataAnalysis01.hs
```

An addition to this line represents work that you contributed to a file but later realized was a mistake. This program will no longer compile with these changes. You may wish that you could remember the contents of the original file. You are in luck. Everything that you have committed to the version control is stored in the repository. Rename your damaged file to `LearningDataAnalysis01Damaged.hs`. We will fix our file back to the last commit:

```
$ git checkout -- LearningDataAnalysis01.hs
```

The `LearningDataAnalysis01.hs` blank file will be added back to your folder. When you inspect your file, you will see that the changes are gone and the file is restored. Hurray!

If you have a project consisting of at least one file, you should use version control. Here is the general workflow for branchless version control:

1. Think.
2. Write some code.
3. Test that code.
4. Commit that code.
5. Go to step 1.

It doesn't take long to see the benefits of version control. Mistakes happen and version control is there to save you. This version control workflow will be sufficient for small projects. Though we will not remind you that you should use version control, you should make a practice of committing your code after each chapter (which is done probably more frequently than this).

Tmux

Tmux is an application that is used to run multiple terminals within a single terminal. A collection of terminals can be detached and reattached to other terminal connections, programs can be kept running in the background to monitor the progress, and the user can be allowed to jump back and forth between terminals, for example, while writing this book, we typically kept tmux running with the following terminals:

- A terminal for the interactive Haskell command line
- A terminal running our favorite text editor while working on the code for a chapter
- A terminal running a text editor with mental notes to ourselves and snippets of code
- A terminal running a text editor containing the text of the chapter we were currently writing
- A terminal running the terminal web browser **elinks** in order to read the Haskell documentation

The prized feature (in our opinion) of tmux is its ability to detach from a terminal (even the one that has lost connection) and reattach itself to the currently connected terminal. Our work environment is a remote virtual private server running Debian Linux. With tmux, we can log in to our server from any computer with an Internet connection and an ssh client, reattach the current tmux session, and return to the testing and writing of the code.

We will begin by installing tmux:

```
$ sudo apt-get install tmux
```

Now, let's start tmux:

```
$ tmux
```

You will see the screen refresh with a new terminal. You are now inside a pseudoterminal. While in this terminal, start the interactive Haskell compiler (`ghci`). At the prompt, perform a calculation. Let's add `2` and `2` by using the prefix manner rather than the typical infix manner (all operators in Haskell are functions that allow for infix evaluation. Here, we call addition as a function):

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
```

```
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
> (+) 2 2
4
```

The interactive Haskell compiler runs continuously. On your keyboard, type *Ctrl* + *B*, followed by *C* (for create). This command creates a new terminal. You can cycle forward through the chain of open terminals by using the *Ctrl* + *B* command, followed by *N* (for next). You now have two terminals running on the same connection.

Imagine that this is being viewed on a remove server. On your keyboard, type *Ctrl* + *B* followed by *D*. The screen will return just prior to you calling tmux. The **[detached]** word will now be seen on the screen. You no longer will be able to see the interactive Haskell compiler, but it will still run in the background of your computer. You can reattach the session to this terminal window by using the following command:

```
$ tmux attach -d
```

Your windows will be restored with all of your applications running and the content on the screen the same as it was when you left it. Cycle through the terminals until you find the Haskell interactive command line (*Ctrl* + *B* followed by *P*, cycles to the previous terminal). The application never stopped running. Once you are finished with your multiplexed session, close the command line in the manner that you normally would (either by using *Ctrl* + *D*, or by typing `exit`). Every terminal that is closed will return you to another open terminal. The tmux service will stop once the last terminal opened within the `tmux` command is closed.

Our first Haskell program

Though this is a book about data analysis using Haskell, it isn't meant to teach you the syntax or features of the Haskell language. What we would like to do for the remainder of the chapter is to get you (the reader) familiar with some of the repeatedly used language features if you aren't familiar with the language. Consider this a crash course in the Haskell language.

The median of a dataset is the value that is present in the middle of the dataset when the dataset is sorted. If there are an even number of elements, the median is the average of the two values closest to the middle of the sorted dataset. Based on this, we can plan an algorithm to compute a median. First, we will sort the numbers. Second, we will determine whether there are an even number of elements or an odd number. Finally, we will return the appropriate middle value.

Create a new folder on your computer where your Haskell files will be stored. You should put all your files in a directory called `~/projects/LearningHaskellDataAnalysis`. Inside this directory, using an editor of your choice, create a file called `LearningDataAnalysis01.hs` (hopefully, you created this file earlier in our demonstration of Git). We will create a module file to store our algorithm to compute the median of a dataset. It will begin with the following lines:

```
module LearningDataAnalysis01 where
import Data.List
```

The first line tells Haskell that this is a module file that contains functions for general usage. The second line tells Haskell that we need the `Data.List` library, which is a part of the Haskell platform. This library contains several versatile functions that are required to use lists, and we will take full advantage of this library.

We will begin by crafting the header of our function:

```
median :: [Double] -> Double
```

The preceding statement states that we have a function named `median` that requires a parameter consisting of a list of floating-point values. It will return a single floating-point value. Now, consider the following code snippet of the `median` function:

```
median :: [Double] -> Double
median [] = 0
median xs = if oddInLength then
              middleValue
            else
              (middleValue + beforeMiddleValue) / 2
  where
    sortedList = sort xs
    oddInLength = 1 == mod (genericLength xs) 2
    middle = floor $ (genericLength xs) / 2
    middleValue = genericIndex sortedList middle
    beforeMiddleValue = genericIndex sortedList (middle-1)
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Haskell interprets our arguments with the help of **pattern matching**. The first two calls under the function header are called **patterns**. Haskell will compare our data to each of these patterns. The first pattern is `[]` (which is an empty list). If our input list to this function is empty, we will return the `0` value. The second pattern is `xs`, which matches a nonempty list. Now, we will evaluate to check whether this list has an odd number of elements or an even number and return the correct value.

The bulk of the work of this function happens under the `where` clause. It is a mistake to think of these statements as a sequential program. These expressions are executed as they are needed to complete the task of the main function. Under the `where` clause, we have five expressions that perform an operation and store a result. We will go over each of them. Consider the first clause:

```
sortedList = sort xs
```

The first clause sorts our list of values and returns the result to `sortedList`. Here, we utilize the `sort` function, which is found in `Data.List`. Consider the second clause:

```
oddInLength = 1 == mod (genericLength xs) 2
```

The second clause determines whether a list has an odd number of elements. We will do this by computing the modulus (using mod) of the length of the list (using `genericLength`) and the number `2`. We will compare this result to the number 1, which must be either true or false. Consider the third clause:

```
middle = floor $ (genericLength xs) / 2
```

The third clause takes the length of the list, divides it by 2, and then computes the mathematical floor of the result. See the `$` operator? This operator tells Haskell that everything on the rest of the line should be treated as a single expression. We could have written this line as `middle = floor ((genericLength xs) / 2)` and it would be valid. This saves us from having to look at an extra set of parentheses. We can take this a step further and use `middle = floor $ genericLength xs / 2` with no parentheses. Readability takes priority over character counting. Consider the fourth and fifth clause:

```
middleValue = genericIndex sortedList middle
beforeMiddleValue = genericIndex sortedList (middle-1)
```

The fourth and fifth clauses use `genericIndex` to pull a specific value from the `sortedList` variable (the first one pulls the value from the computed middle and the second pulls it from the element that is immediately before the middle). The fifth clause has a potential problem—on a list with one element, the middle element is 0 and the element before the middle element is -1.

If you recall our discussion earlier on lazy evaluation, none of these statements are called unless needed. Back in the main portion of the function, you can see the description of `median`. The same can be seen in the following:

```
median xs = if oddInLength then
              middleValue
            else
              (middleValue + beforeMiddleValue) / 2
```

Consider an example of a list with one element. The first expression that is encountered in our list of `where` clauses will be `oddInLength` (since this is evaluated in the `if` statement). The `oddInLength` expression should evaluate to true. Thus, we execute the true branch of the conditional expression. The `middleValue` expression requires you to call `genericIndex` function on `sortedList` (which executes the two remaining `where` clauses). In this example, `beforeMiddleValue` will not be executed.

We will build a wrapper program that utilizes our `median` function call. Create a second file called `median.hs`, which will serve as our wrapper to the module:

```
import System.Environment (getArgs)
import LearningDataAnalysis01
main :: IO ()
main = do
  values <- getArgs
  print . median $ map read values
```

You can see that the last line of the file states—*take all the values from the command line, read them as Double values, pass them to the median, and print the result*. You might ask yourself how Haskell knows how to read these values as `Double` values and not anything else. This is where the magic of type inference happens. Because our `median` function specified that it requires a list of `Double` values, this information is passed on to `read` to make sure that the information is interpreted as a `Double` type. The `map` function makes sure that `read` is applied to every element in values. Finally, the `print` function prints the result.

Let's compile your new program from the command line. We are ready to test:

```
$ ghc median.hs -o median
```

Great. We will have a new executable in our directory called `median`. From the command line, test a few values:

```
$ ./median
0.0
$ ./median 1
1.0
$ ./median 1 2
1.5
```

```
$ ./median 2 1
1.5
$ ./median 1 2 3
2.0
$ ./median 2 3 1
2.0
$ ./median 3 4 5 1 2
3.0
```

From this small sample of input, we believe that our function is working correctly. We can use this function on later datasets to find the median of samples.

Interactive Haskell

This section will be used to familiarize you with the Haskell interactive command line. Before we introduce you to the interactive command line, we will introduce the optional configuration file that you can create in `~/.ghci` in your home folder. We have configured ours with the following code:

```
:set prompt "> "
```

The preceding code tells the interactive command line to display a single `>` as the prompt. You can start the interactive command line using the `ghci` command. Here is what you will see when the command line is started:

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
>
```

You can execute simple equations using either the familiar infix notation, or the functional notation:

```
> 2 + 2
4
> (+) 2 2
4
> 2 + 4 * 5
22
> (+) 2 $ (*) 4 5
22
```

Note that we need to use `$` here in order to tell Haskell that the `(*) 4 5` multiplication portion is an argument to the `(+) 2` addition portion.

An introductory problem

This introductory problem will serve as a way of explaining the features of the Haskell language that are used repeatedly in this book. The problem is that we wish to know the location of the vowel characters of a given word, for example, in the word `apple`, there are two vowels (the first and fifth letters). Given a string `apple`, we should return a list of [1, 5]. We will go through the thought process of solving this problem and turn our solution into a function. You can use the `elemIndices` function that can be found in the `Data.List` module, but we will chose not to do so for teaching purposes.

First, we will declare a variable to store our word. In this example, we will use the word `apple`:

```
> let word = "apple"
```

We will assign a number to each letter in our word using `zip` and an infinite list of numbers. The `zip` function will perform a pair-wise merge of two lists to create a list of tuples. A **tuple** is a type of list structure that can store types in a heterogeneous manner. In the following code, we will combine the integer and character types:

```
> zip [1..] word
[(1,'a'),(2,'p'),(3,'p'),(4,'l'),(5,'e')]
```

The expression `[1..]` is an infinite list of numbers. If you type this in the interactive command line, numbers will appear until you decide to stop it. By using it in conjunction with `zip`, we only take what we need. There are five letters in `apple`. So, we only take five elements from our infinite list. This is an example of lazy evaluation at work.

Next, we will filter our list to remove anything that is not a vowel character. We will do this with the help of the `filter` function, which requires us to pass a lambda function with the rule that defines what is allowed in the list of values:

```
> filter (\(_, letter) -> elem letter "aeiouAEIOU") $ zip [1..] word
[(1,'a'),(5,'e')]
```

Let's take a closer look at the lambda expression that is within the parentheses that begin with `(\` and end with `)`. Using the `:t` option, we can inspect how Haskell interprets this function:

```
> :t \(_, letter) -> elem letter "aeiouAEIOU"
\(_, letter) -> elem letter "aeiouAEIOU" :: (t, Char) -> Bool
```

The function requires a pair of values. The first value in the pair is identified with _, which indicates that this is a wild card type. You can see that Haskell identifies it with the `t` generic type. The second value in the pair is identified by `letter`, which represents a character in our string. We never defined that letter was a `Char` type, but Haskell was able to use type inference to realize that we were using the value in a list to search for the value among a list of characters and thus, this must be a character. This lambda expression calls the `elem` function, which is a part of the `Data.List` module. The `elem` function returns a `Bool` type. So, the return type of `Bool` is also inferred. The `elem` function returns true if a value exists in a list. Otherwise, it returns false.

We need to remove the letters from our list of values and return a list of only the numbers:

```
> map fst . filter (\(_, letter) -> elem letter "aeiouAEIOU") $ zip [1..]
word
[1,5]
```

The `map` function, like the filter, requires a function and a list. Here, the function is `fst` and the list is provided by the value returned by the call to the filter. Typically, tuples consist of two values (but this is not always the case). The `fst` and `snd` functions will extract the first and second values of a tuple, as follows:

```
> :t fst
fst :: (a, b) -> a
> :t snd
snd :: (a, b) -> b
> fst (1, 'a')
1
> snd (1, 'a')
'a'
```

We will add our newly crafted expression to the `LearningDataAnalysis01` module. Now, open the file and add the new function towards the end of this file using the following code:

```
-- Finds the indices of every vowel in a word.
vowelIndices :: String -> [Integer]
vowelIndices word =
  map fst $ filter (\(_, letter) -> elem letter "aeiouAEIOU") $
  zip [1..] word
```

Then, return to the Haskell command line and load the module using `:l`:

```
> :l LearningDataAnalysis01
[1 of 1] Compiling LearningDataAnalysis01 ( LearningDataAnalysis01.hs,
interpreted )
Ok, modules loaded: LearningDataAnalysis01.
```

In the next few chapters, we will clip the output of the `load` command.
Your functions are now loaded and ready for use on the command line:

```
> vowelIndices "apple"
[1,5]
> vowelIndices "orange"
[1,3,6]
> vowelIndices "grapes"
[3,5]
> vowelIndices "supercalifragilisticexpialidocious"
[2,4,7,9,12,14,16,19,21,24,25,27,29,31,32,33]
> vowelIndices "why"
[]
```

You can also use the `median` function that we used earlier. In the following code, we will pass every integer returned by `vowelIndices` through `fromIntegral` to convert it to a `Double` type:

```
> median . map fromIntegral $ vowelIndices
"supercalifragilisticexpialidocious"
20.0
```

If you make changes to your module, you can quickly reload the module in the interactive command line by using `:r`. This advice comes with a warning—every time you load or reload a library in Haskell, the entire environment (and all your delicately typed expressions) will be reset. *You will lose everything on doing this*. This is typically countered by having a separate text editor open where you can type out all your Haskell commands and paste them in the GHCi interpreter.

Summary

This chapter looked at Haskell from the perspective of a data analyst. We looked at Haskell's feature set (functional, type-inferred, and lazy). We saw how each of these features benefit a data analyst. We also spent some time getting acquainted with our environment, which includes the setting up of Haskell, cabal, Git, and tmux. We ended the chapter with a simple program that computes the median of a list of values and creates a function to find the vowels in a string.

2
Getting Our Feet Wet

This chapter looks at Haskell's type system by examining where it works in your favor as well as the common obstacles that you may be face when trying to understand it. We will also work with csv files, a common format that is used to store datasets. The csv file type is cumbersome to work with. So, we will spend the remainder of this chapter in learning how to convert csv files into SQLite3 databases.

In this chapter, we will cover the following:

- Type is king—the implications of strict types in Haskell
- Working with csv files
- Converting csv files to the SQLite3 format

Type is king – the implications of strict types in Haskell

Haskell is a language that prides itself with regard to the correctness and conciseness of the language, and the robust collection of libraries via which you can explore more while maintaining the purity of a purely functional programming language. For those who are new to Haskell, there are a number of innovative features. Those coming from the world of C-style programming languages will admire Haskell's type inference capabilities. A language that's both strongly typed and type-inferred during compile time is a welcome change. A variable can be assigned a type once and passed through a variety of functions without you ever having to be reminded of the type. Should the analyst use the variable in a context that is inappropriate for the assigned type, it can be flagged accordingly during compile time rather than run time. This is a blessing for data analysts. The analyst gets the benefits of a statically typed language without having to constantly remind the compiler of the types that are currently in play. Once the variable types are set, they are never going to change. Haskell will only change the structure of a variable with your permission.

The flip side of our strictly typed code is quickly encountered in the study of data analysis. Several popular languages will convert an integer to a rational number if the two are used in an expression together. Languages typically do this because we, as humans, typically think of numbers as just numbers, and if an expression works mathematically, it should work in an expression in a programming language. This is not always the case in Haskell. Some types (as we will see in the following section) must be explicitly converted if an analyst wants them to be used in a context that Haskell deems potentially unsafe.

Computing the mean of a list

Allow us to demonstrate this problem with an example, which also serves as our first step into the world of data analysis problems. The **mean** (or the **average**) of a list of values is considered a summary statistic. This means that it allows us to express lots of information with a single value. Because of the ease with which the mean can be calculated, it is one of the most frequently used (and misused) summary statistics. According to the United States Census Bureau, the average sale price of a new home in the United States in 2010 was $272,900. If you are familiar with home values in the United States, this value might seem high to you. The mean of a dataset is easily skewed by outlier information. In the context of home prices, there are a few, rare new homes that were sold that were worth more than $125 million. This high home price will shift the mean away from the *middle* concept that is generally believed to be represented by the mean. Let us begin by computing the mean of a list in Haskell. The mean of a list of numbers is computed by finding the summation of this list and dividing this sum by the number of elements in the list. The data presented here represents the final score made by the Atlanta Falcons in each of their games during the 2013 NFL football season. Not a football fan? Don't worry. Focus only on the numbers. The purpose of this example is to make you work on a small, real-world dataset. There are 16 games in this dataset, and we can present them all in a single line, as follows:

```
> let falconsScores = [17,31,23,23,28,31,13,10,10,28,13,34,21,27,24,20]
```

Computing the sum of a list

At this point, `falconsScores` is just a list of numbers. We will compute the sum of these values. The Prelude package consists of a handful of functions that are ready for use in the **GHC** environment. There is no need to import this library. These functions work right out of the box (so to speak). Two of these functions are `sum` and `length`:

```
> let sumOfFalconsScores = sum falconsScores
> sumOfFalconsScores
353
```

The `sum` function in the Prelude package does what you may expect; (from the Haskell documentation), the `sum` function computes the sum of a finite list of numbers.

Computing the length of a list

So far, so good. Next, we need the length of the list. We will find it out with the help of the following code:

```
> let numberOfFalconsGames = length falconsScores
> numberOfFalconsGames
16
```

The `length` function also does what you may expect. It returns the length of a finite list as an `Int`.

Attempting to compute the mean results in an error

In order to compute the mean score of the 2013 season of the Atlanta Falcons, we will divide the sum of the scores by the number of scores, as follows:

```
> let meanScoreofFalcons = sumOfFalconsScores / numberOfFalconsGames
<interactive>:82:61:
    Couldn't match expected type 'Integer' with actual type 'Int'
    In the second argument of '(/)', namely 'numberOfFalconsGames'
    In the expression: sumOfFalconsScores / numberOfFalconsGames
    In an equation for 'meanScoreofFalcons':
        meanScoreofFalcons = sumOfFalconsScores / numberOfFalconsGames
```

Oh dear. What is this? I assure you that Haskell can handle simple division.

Introducing the Fractional class

Since mathematical division involves fractions of numbers, Haskell requires us to use a `Fractional` type when dividing numbers. We will inspect our data types, as follows:

```
> :t sumOfFalconsScores
sumOfFalconsScores :: Integer
> :t numberOfFalconsGames
numberOfFalconsGames :: Int
```

The `sum` function returned an integer based on the data (some versions of Haskell will return the more generic `Num` class in this instance). `Integer` in Haskell is an arbitrary precision data type. We never specified a data type. So, Haskell inferred the `Integer` type based on the data. The `length` function returns data as an `Int` type. Not to be confused with `Integer`, `Int` is a bounded type with a maximum and minimum bound to the values of this type. Despite the similar use of both types, they have some important differences (`Integer` is unbounded, `Int` is bounded). Using `Integer` and `Int` together has a potential to fail at runtime. Instead of failing at runtime, the compiler notices this potential to fail and flags it during the compile time.

The fromIntegral and realToFrac functions

The `fromIntegral` function is our primary tool for converting integral data to the more generic `Num` class. Since our second operand (`numberOfFalconsGames`) is of the `Int` type, we can use `fromIntegral` to convert this from `Int` to `Num`. Our first operand is of `Integer` type and the `fromIntegral` function will work in this circumstance as well, but we should avoid that temptation (if this list consisted of floating-point numbers, `fromIntegral` would not work). Instead, we should use the `realToFrac` function, which converts (as the name implies) any numerical class that extends the `Real` type to a `Fractional` type on which the division operator depends and can hold an unbounded integer:

```
> let meanFalconsScore = (realToFrac sumOfFalconsScores) / (fromIntegral
numberOfFalconsGames)
> meanFalconsScore
22.0625
> :t meanFalconsScore
meanFalconsScore :: Double
```

Creating our average function

Now that we have enjoyed exploring the Haskell type system, we should probably build an `average` function. We see that the type system automatically recognizes the types in our function and that we are converting a list of `Real` types to `Fractional`. Here is our `average` function:

```
> let average xs = realToFrac (sum xs) / fromIntegral (length xs)
> :t average :: (Fractional a, Real a1) => [a1] -> a
```

We see from the type description that type `a` is a `Fractional` type (our output) and that type `a1` is a `Real` type (our input). This function should support integers, floating-point values, and mixtures of integer and floating point values. Haskell isn't being creative with automatically generated type names such as `a` and `a1`, but they will do. As always, we should test:

```
> average [1, 2, 3]
2.0
> average [1, 2, 3.5]
2.1666666666666665
> average [1.5, 2.5, 3.5]
2.5
> let a = [1, 2, 3]
> average a
2.0
> average []
NaN
```

The final test in this list reports that the average empty list is **NaN** (short for **Not A Number**).

The genericLength function

Things appear to be in working order. There is an additional way of finding the length of the list that is found in the `Data.List` package and is called `genericLength`. The `genericLength` function does the same as the `length` function, but with the added effect that the return value is `Num` and will happily work with the division operator without converting the value. Testing to see if this new version of `average` is working will be left as an exercise to the reader:

```
> import Data.List
> let average xs = realToFrac(sum xs) / genericLength xs
> :t average
average :: (Fractional a, Real b) => [b] -> a
```

We should add our new function to average a list of numbers to the `LearningDataAnalysis02.hs` module:

```
module LearningDataAnalysis02 where
-- Compute the average of a list of values
average :: (Real a, Fractional b) => [a] -> b
average xs = realToFrac(sum xs) / fromIntegral(length xs)
```

As a data analyst, you will be working with various forms of quantitative and qualitative data. It is essential that you understand how the typing system in Haskell reacts to data. You will be responsible for explicitly saying that a data type needs to change should the context of your analysis change.

Metadata is just as important as data

Data comes in all sizes, from the small sets that are typed on a command line to the large sets that require data warehouses. Data will also come in a variety of formats: spreadsheets, unstructured text files, structured text files, databases, and more. Should you be working for an organization and responsible for analyzing data, datasets will come to you from your management. Your first task will be to figure out the format of the data (and the necessary software required to interact with that data), an overview of that data, and how quickly the dataset becomes outdated. One of my first data projects was to manage a database of United States Civil War soldier death records. This is an example of a dataset that is useful to historians and the families of those who served in this war, but also one that is not growing. Your author believes that the most interesting datasets are the ones that are continually growing and changing. A helpful metadata document that accompanies the dataset will answer these questions for you. These documents do not always exist. So pull your manager over and get him or her to answer all of these questions and create the first metadata document of this dataset.

Working with csv files

A common format to exchange data externally between an organization and the outside world is the **CSV** (short for **Comma Separated Values**) format. Files of this format have the `.csv` extension. Most spreadsheet software packages will be able to import and export csv files. Also, many database software packages will have the ability to export tables to csv files. The format of a csv file is relatively simple. A csv file is a plain text document containing multiple records, and each record begins on a new line (but it will not necessarily end on the same line). The first line of the document is not a record, but it contains a description of each column in the dataset. Each record will consist of values (or fields), each separated by a delimiter character (usually a comma). A field that is wrapped in double quotes may contain delimiter characters and newline characters. There is no documented standard that defines a proper csv file. The first line in the document might be the first record of the data. A record can span multiple lines. Some people like using tabs as their delimiter. Despite being such a deceptively simple file format, there is plenty of ambiguity. It is good practice to avoid writing your own csv library and to instead use a trusted library.

Preparing our environment

Using the cabal tool, we need to download the `Text.CSV` library. From the command line, type the following:

```
cabal install csv
```

We wish to use the `Text.CSV` library to parse our csv file. It will be helpful to learn a little more about this library. There are three Haskell types that are found in `Text.CSV`. The first type is `CSV`, which is our representation of a csv file. A `CSV` is composed of a list of the `Record` type (the second type). Each `Record` is a list of the `Field` type (the third type). Each `Field` is a `String`. Internally, a csv file is represented using these types, but you can think of a csv file as an expression of the `[[String]]` type.

Throughout our discussion on csv files, I will refer to the csv file format as `csv` and the Haskell CSV expression type as `CSV`.

Before we continue any further, we will perform some housekeeping on our `LearningDataAnalysis02.hs` file. These import statements will be necessary for our work in the following example. Any time we mention an import statement, add it to the beginning of the file after the module line:

```
import Data.List
import Data.Either
import Text.CSV
```

Describing our needs

Having addressed the ambiguity of csv files, when csv files are addressed in this book, we will assume the following:

- The first line in the file contains a description of each column
- Each line after the first line contains exactly one record
- Each record field is separated by a comma character
- The datasets may have quoted strings, which will not contain newline characters

We should get some practice with regard to performing simple data analysis-related work on a csv file. The **United States Geological Survey** (**USGS**) presents a variety of data to the general public, including recent data on worldwide earthquakes. By visiting the following site, you can download earthquake-related data that spans the past hour, day, seven days, and 30 days:

```
http://earthquake.usgs.gov/earthquakes/feed/v1.0/csv.php
```

From this site, you will see that the format of the data is in csv, the columns are clearly defined on the first line of every csv file, and the data is updated every five minutes.

I downloaded the csv file representing all the earthquakes in the past week. When the file was downloaded, the file size was just over 220 KB in size and contained 1,412 records of earthquakes. Based on our discussion in the previous section, we will attempt to answer the question—*What was the average magnitude of every earthquake in the past week?*

Crafting our solution

Before we consider an answer to our question, we must consider how we are going to access the data. Our data is buried in a file and we need to extract this information. We will craft a function that applies a desired operation (which will be supplied by the user) to every value in a single column of a csv file. We will make this function generic with regard to a desired operation because we wish to make it possible for users to pass it in their own functions (not just the function used in this example). Our goal is to craft a function that accepts the following input parameters:

- A function that will perform our desired operation (in this case, the desired operation will be the average function written earlier)
- A filename representing a csv file (a `FilePath`)
- A column within our csv file on which the desired operation will be applied (a `String`)

In addition, our function should return a value, and the type of this value type will be determined by the desired operation's function (which will require a polymorphic type—more on this will be discussed later). Because each field in our file is a string, the input type of the passed function must also be a `String`. Also, we need some mechanism to handle the errors that might arise while performing our work. For this, we will return our values wrapped in the `Either` type, which is commonly used to pass error messages. Finally, since this is going to involve file I/O, we need to wrap the `Either` type in the `IO` monad.

Having taken all this into consideration, we will now present the signature of this function:

```
applyToColumnInCSVFile :: ([String] -> b) -> FilePath -> String -> IO
  (Either String b)
```

This is a lot to take in and we should not expect those new to data analysis in Haskell to think of all these considerations immediately. We are going to put this aside and come back to it later. There are two design considerations that will go into this function:

- Finding the column index of the specified column that interacts with the csv file itself
- In typical Haskell fashion, these concepts will be demonstrated with their own functions

Finding the column index of the specified column

We will start with the easier of the two design considerations—finding the column index of the specified column. This function will accept a CSV value and a column name as input, and it should return either the integer of the column or an error message. To handle the returning of the two possible types, we will use the `Data.Either` library. Since this function will take a CSV value (rather than a CSV filename), we do not have to concern ourselves with the wrapping of the returned value in the `IO` monad. Here is our function:

```
getColumnInCSV :: CSV -> String -> Either String Integer
getColumnInCSV csv columnName =
  case lookupResponse of
    Nothing -> Left
      "The column does not exist in this CSV file."
    Just x -> Right (fromIntegral x)
  where
  -- This line looks to see if column is in our CSV
  lookupResponse = findIndex (== columnName) (head csv)
```

The heart of this function is the very last line, in which we called the `findIndex` function using the column name and the first line of the csv file as the input. The `findIndex` function is found in `Data.List`, and it will return a `Maybe Int` type constructor. `Maybe` is one of the basic Haskell types consisting of two type constructors—`Just` and `Nothing`. On finding the column, `findIndex` will return a `Just` type, but if the column is not found, then `Nothing` is returned. We will perform pattern matching on the returned `Maybe` type to convert this to an `Either` type. On finding `Nothing`, we will convert this pattern into an error message that explicitly states that *The column does not exist in this csv file*. On finding the specific index of the column, we will return that column index and convert it into an `Integer` in the process.

We will perform some demonstrations of our function. For this demonstration, we will use the earthquake csv file. Our precondition for this test is that the csv file exists and is properly formatted. What we do not assume is that the desired column will exist in the csv file. In this example, we know there exists a column called `mag`. Our function resides in a module file called `LearningDataAnalysis02.hs`. We will load this module in the interactive interpreter using the `:l` (for load) command, as follows:

```
> :l LearningDataAnalysis02.hs
> csv <- parseCSVFromFile "all_week.csv"
> either (\error -> Left "Problem Reading File") (\csv -> getColumnInCSV
csv "mag") csv
Right 4
```

Good. Note that the column exists on index 4. (Haskell, like C, C++, and Java, begins list indices with 0) The `Right` wrapper identifies that the function correctly identified the column and this column is `4`. We will test for something that fails, as follows:

```
> either (\error -> Left "Problem Reading File") (\csv -> getColumnInCSV
csv "not a column") csv
> Left "The column does not exist in this CSV file."
```

There is no column named `not a column` in this file and our function correctly reports this using the `Left` wrapper. In line with the typical use of `Either`, `Right` is used for the successful execution and `Left` is to report error messages.

The Maybe and Either monads

If you are new to Haskell, you might ask yourself, "What are Maybe, Just, and Nothing?" `Maybe` is a data type with two constructors—`Just` and `Nothing`. The `Just` constructor takes one value and `Nothing` takes no values. `Maybe` is also an instance of a general and powerful abstraction called a monad. In order to explain the importance of `Maybe`, I'm going to describe the features from other programming languages. Imagine the `int` data type found in the C language. C's `int` and Haskell's `Int` share a few things in common. Both are bounded types with a maximum and minimum value and are only allowed to hold the integer data, and both must always have a value associated with the variable of this type. In C, if a variable of the `int` type isn't given a value, the initial value will be whatever the compiler decides it to be (which is why you should always give C variables an initial value). It still has a value! In fact, there's no way to express that a C `int` variable doesn't have any value. Contrast this with the language of Java, which allows you to create an object of the `Integer` class. Java will initialize the variable with the `null` value. Java's `null` specifically means that the variable has no value, which means that all Java references can have either a value of the specified type or none at all. Haskell allows us to do the same thing using `Maybe`. When a variable is defined as a `Maybe` type, it means something similar to Java's notion of a reference. It is an explicit way of saying that this variable might hold a value of the specified data (the `Just` value) or it might hold `Nothing`.

Likewise, you might be asking yourself about the `Either` construct that we keep using. `Either` is a more advanced form of `Maybe`. When we encounter a variable in Haskell representing a `Maybe`, we're saying that it maybe holds a value, but we're not saying why if you find `Nothing`. The `Either` construct was created to address this. An `Either` expression comprises of two expressions—`Left a` and `Right b`. `Either` allows us to represent an expression that can evaluate two different expressions, each with their own constructors (`Left` and `Right`). Often, the `Either` expression is used to report error messages in code since computations can either be successful or can return an error message on failure. When using the `Either` expression, we can define the `Left` and `Right` expressions with whatever we want, but by convention, the `Right` side of `Either` is used to return the results of a successful execution, and the `Left` side of `Either` is used to return a `String` error message outlining why a function call failed. To help you remember, try remembering that `Right` is right and `Left` is wrong.

Applying a function to a specified column

Our next function will do the actual work of applying our desired function to a column within the csv file. This will be similar to the primary function with the only difference being that this function will assume that an already opened CSV data expression will be passed instead of a filename:

```
applyToColumnInCSV :: ([String] -> b) -> CSV -> String -> Either
  String b
applyToColumnInCSV func csv column = either
    Left
    Right . func . elements
  columnIndex
  where
  columnIndex = getColumnInCSV csv column
  nfieldsInFile = length $ head csv
  records = tail $
    filter (\record -> nfieldsInFile == length record) csv
  elements ci = map
    (\record -> genericIndex record ci) records
```

Note that this function signature requests a value of CSV expression instead of `String`. This allows us to focus on the task of data analysis instead of being concerned about the file's I/O. There are a few considerations here that need to be explained. Starting with the `where` clause, we used the `getColumnInCSV` function to get the column index. Next, we counted the number of fields represented on the first row of our csv file. We used the number of fields to filter the records and ignore any records that do not have one field per field heading. The line beginning with `elements ci` will take the records and reduce them to a single column defined by `columnIndex`. Since `columnIndex` represents either a real column index or an error message, our function will make sure that the error message is propagated back out in a `Left` call or the actual work of applying a function to our column is performed inside a `Right` call.

We need to return to the discussion of the polymorphic type used in our function call identified as `b`. We will apply a function to a column in the csv file, but no assumptions are made with regard to the type of data that exists in the column or the function passed. In the context of our problem, we wish to compute the average magnitude of earthquakes over the span of a week. The data in the column is the `String` data that needs to first be interpreted into the `Double` value. Here is a simple helper function to do just that. Both this function and the `average` function need to be in `LearningDataAnalysis02.hs`:

```
readColumn :: [String] -> [Double]
readColumn xs = map read
```

As always, we should test our function in the following manner:

```
> csv <- parseCSVFromFile "all_week.csv"
> either
 (\error -> Left "Problem Reading File")
 (\csv -> applyToColumnInCSV (average . readColumn) csv "mag")
 csv
Right 1.6950637393767727
```

Note that the average magnitude for all earthquakes over the course of a sample week is 1.70 (rounded). Of course, if you are following along, your result will vary. We need to test this function again with a bad column name, as follows:

```
> either
 (\error -> Left "Problem Reading File")
 (\csv -> applyToColumnInCSV (average . readColumn) csv "not a column")
 csv
Left "The column does not exist in this CSV file."
```

This function appears to be in working order. Now, we will return to the writing of a function that puts what we have learned together. This function will be responsible for the opening of a csv file and the calling of the helper functions, as follows:

```
-- Opens a CSV file and applies a function to a column
-- Returns Either Error Message or the function result
applyToColumnInCSVFile ::
 ([String] -> b) -> FilePath -> String -> IO (Either String b)
applyToColumnInCSVFile func inFileName column = do
  -- Open and read the CSV file
  input <- readFile inFileName
  let records = parseCSV inFileName input
  -- Check to make sure this is a good csv file
  return $ either
    handleCSVError
    (\csv -> applyToColumnInCSV func csv column)
    records
  where
    handleCSVError ==
      Left "This does not appear to be a CSV file."
```

The only consideration that needs to be addressed in this function is the use of the `return` call. This call makes sure that the `Either` value is wrapped in an `IO` monad.

Now, we will test the following code:

```
> :l LearningDataAnalysis02.hs
> applyToColumnInCSVFile (average . readColumn) "all_week.csv" "mag"
Right 1.6950637393767727
```

We will test the preceding code again, but with a nonexistent column, as follows:

```
> applyToColumnInCSVFile (average . readColumn) "all_week.csv" "not a
column"
Left "The column does not exist in this CSV file."
```

It takes a lot of work in Haskell to reach such a simple function call and see a result. Once you build the layers of resources in your modules, you will see us pick up speed, for example, in the `Data.List` module, there are the `maximum` and `minimum` functions that we can use immediately with the work we have done so far, as follows:

```
> applyToColumnInCSVFile (maximum . readColumn)
                         "all_week.csv" "mag"
Right 6.7
> applyToColumnInCSVFile (minimum . readColumn)
                         "all_week.csv" "mag"
Right (-0.3)
```

While writing this example, I learned that the Richter magnitude scale can record negative values. I readily admit that I am not a domain expert in the geological sciences. Knowledge of the domain in which you will be working is something that is essential to an accurate interpretation of the data that you will be analyzing. When it comes to the geological sciences (or any other unfamiliar domain), should you lack the same expertise as me, do not be shy about partnering with an expert-level friend.

Converting csv files to the SQLite3 format

To speed up our venture in data analysis, we will convert our csv file into a SQLite3 database. With a SQLite3 database, our work becomes easier. We spend less time on lines of code that are related to the integrity of the data and more time filtering, splicing, and combining columns of data to meet our needs.

We wish to craft a function that will produce a SQLite3 database based on the contents of a csv file. SQLite3 is a relational database management system. In csv files, we explore data through records, where every field in a record has a column heading. In relational databases, we see much of the same approach but with a few important differences. Relational databases comprise of one or more tables, each identified by a name. Each table comprises of zero or more records, where each record comprises of one field per column heading, making a table analogous to a csv file. An important distinction between a table and a csv file is that tables require each field to maintain adherence to a data type. On the contrary, csv files treat all fields as text.

Preparing our environment

To begin preparing our environment, make sure that you have SQLite3 and the **Haskell Database Connectivity** (**HDBC**) package installed on your system. The HDBC is known as a database abstraction layer. We can interface directly with SQLite3, but then, we will be attached to SQLite3 in all our code. By using an abstraction layer, we can continue using SQLite3 or change to a different relational database system in the future (such as MySQL or PostgreSQL) with minimal code changes. Within Haskell, we are going to need the HDBC package, the `sqlite` package, and the `HDBC-sqlite3` package. On Debian-based systems, this can be done with `apt-get` on the command line. We can get the two packages on our system using `apt-get`, as follows:

```
sudo apt-get install sqlite3
sudo apt-get install libghc-hdbc-sqlite3-dev
```

To install all of the necessary modules, we can run the following `cabal` install line:

```
cabal install HDBC sqlite HDBC-sqlite3
```

Got all of this installed? Good. As with the previous example, we will break the function into smaller functions, each with their own responsibility. In this example, we will craft a function to open the csv file (which is similar to the function used in the previous example) and a second function to create the database based on a CSV expression.

Those of you who are using Windows can also participate, but you will need to download and install the SQLite binaries from `http://www.sqlite.org`. There, you will find the `Sqlite` and `HDBC-sqlite` packages. These can be hooked during the installation using the `--extra-lib-dirs and --extra-include-dirs` flags.

Describing our needs

To craft our function to create a SQLite3 database from a CSV value, we will require the desired table name, the desired filename for the SQLite3 file, and the type information for each column in the CSV value. For each column in our table, we can take the same column name from the CSV value, but this presents a problem. Often, the column names used in csv files contain spaces and symbols, neither of which are handled elegantly by database systems or the programmers and analysts who use them. An option that you have is the manual editing of the csv file with the desired column names. While this is a solution, our dataset immediately becomes less portable. To transfer our programs to someone else, we will have to request that they also manually edit any new data sources to the desired column names. I humbly request that you resist any temptation to manually edit your data sources except when you have to correct something nuanced that cannot be automated.

Inspecting column information

The USGS has an excellent glossary describing each of the columns in their datasets, including plain English definitions, data types, and numerical ranges. Each of their columns are represented by single words (or short phrases using camel case). We have no need to rename the columns in this dataset.

Let us examine the column headings used in the earthquake dataset. Using the glossary information found at `http://earthquake.usgs.gov/earthquakes/feed/v1.0/glossary.php`, I included the data type information, which is as follows:

- `time`: ISO 8601 Timestamp String
- `latitude`: Decimal
- `longitude`: Decimal
- `depth`: Decimal
- `mag`: Decimal
- `magType`: String
- `nst`: Integer
- `gap`: Decimal
- `dmin`: Decimal
- `rms`: Decimal
- `net`: Decimal
- `id`: String
- `updated`: ISO 8601 Timestamp String
- `place`: String
- `type`: String

There are four data types used in the dataset—Integer, Decimal, String, and ISO 8601 Timestamp Strings (which can be used by the time and date functions in SQLite3). Each data type must be converted to its equivalent type in SQLite3. Fortunately, these conversions are going to be relatively straight forward and can be listed as follows:

- Integer will become `INTEGER`
- Decimal will become `REAL`
- String will become `TEXT`
- ISO 8601 Timestamp String will become `TEXT`

Before we get much further, we need to include our import statements for our database libraries, which are as follows:

```
> import Database.HDBC
> import Database.HDBC.Sqlite3
```

Crafting our functions

Here is the function that turns a CSV expression into a database. Rather than return anything, we print confirmation messages on the screen, as follows:

```
-- Converts a CSV expression into an SQL database
-- Returns "Successful" if successful,
-- error message otherwise.
convertCSVToSQL ::
    String -> FilePath -> [String] -> CSV -> IO ()
convertCSVToSQL tableName outFileName fields records =
  -- Check to make sure that the number of
  --   columns matches the number of fields
    if nfieldsInFile == nfieldsInFields then do
    -- Open a connection
    conn <- connectSqlite3 outFileName

    -- Create a new table
    run conn createStatement []
    -- Load contents of CSV file into table
    stmt <- prepare conn insertStatement
    executeMany stmt (tail
      (filter (\record -> nfieldsInFile == length record)
      sqlRecords))

    -- Commit changes
    commit conn
    -- Close the connection
    disconnect conn
    -- Report that we were successful
    putStrLn "Successful"
    else
    putStrLn
    "The number of input fields differ from the csv file."
```

```
where
  nfieldsInFile = length $ head records
  nfieldsInFields = length fields
  createStatement = "CREATE TABLE " ++
                    tableName ++
                    " (" ++ (intercalate ", " fields) ++")"
  insertStatement = "INSERT INTO " ++
                    tableName ++"++" VALUES (" ++
                    (intercalate ", "
                    (replicate nfieldsInFile "?")) ++ ")""?"))
                    ++ ")"
  sqlRecords = map (\record ->
               map (\element -> toSql element)
               record
               ) records
```

There's a lot to explain in the above code. Here are the related HDBC statements:

- The `connectSqlite3` statement will create a blank SQLite3 file. The `run conn createStatement []` statement creates an initial table within the database. Inspecting the `createStatement` line in the `where` clause reveals that this is a simple SQL statement to create a table based on the types and column names supplied by the `fields` input expression. The `run` statement allows us to perform string interpolation on SQL statements by replacing the `?` symbols with values supplied in a list. We did not use this particular feature in the `connectSqlite3` statement, but we must still pass an empty list.
- The `stmt <- prepare conn insertStatement` statement prepares the `INSERT SQL` statement to batch process of each record.
- The `executeMany stmt` statement performs the batch processing of SQL statements for each record in the CSV expression with the number of columns equal to the number of column headings. If there is a record in your CSV expression that has too many or too few fields, it is going to be skipped.

- The `commit conn` statement does just what its name suggests. It commits the recent changes to the file.
- The `disconnect conn` statement formally disconnects from the database (that is, it closes the file).
- In the `where` clause, you may notice that all the fields in the database are converted to their respective SQL type using the `toSql` function.

Here is the `convertCSVFileToSQL` function that will open a csv file and pass the contents of our file to the `convertCSVToSQL` function. Since this is functionally similar to the `applyToColumnInCSVFile` function mentioned earlier, we will spare you the details of how it works. We will also test the two functions introduced in this section together:

```
-- Converts a CSV file to an SQL database file
-- Prints "Successful" if successful, error message otherwise
convertCSVFileToSQL ::
    String -> String -> String -> [String] -> IO ()
convertCSVFileToSQL inFileName outFileName tableName fields = do
    -- Open and read the CSV file
    input <- readFile inFileName
    let records = parseCSV inFileName input

    -- Check to make sure this is a good csv file
    either handleCSVError convertTool records
 where
    convertTool = convertCSVToSQL tableName outFileName fields
    handleCSVError csv =
        putStrLn "This does not appear to be a CSV file."
```

We will begin our testing, as follows:

```
> :l LearningDataAnalysis02.hs
> convertCSVFileToSQL "all_week.csv" "earthquakes.sql" "oneWeek" ["time
TEXT", "latitude REAL", "longitude REAL", "depth REAL", "mag REAL",
"magType TEXT", "nst INTEGER", "gap REAL", "dmin REAL", "rms REAL", "net
REAL", "id TEXT", "updated TEXT", "place TEXT", "type TEXT"]
Successful
```

At the completion of this step, a new file will exist in your current working directory with the `earthquakes.sql` filename. We will inspect our new database, as follows:

```
> conn <- connectSqlite3 "earthquakes.sql"
> magnitudes <- quickQuery' conn "SELECT mag FROM oneWeek" []
> :t magnitudes
magnitudes :: [[SqlValue]]
```

Note that the information of the type here is based on a simple SQL statement to our database. The data is returned as a list of `SqlValue`. We will inspect the first element that was returned by converting the data into `Double` by using the `safeFromSql` function, as follows:

```
> fromSql $ head $ head magnitudes :: Double
0.5
```

To end this chapter, let's apply the data returned by the SQL statement to the `average` function that was written at the beginning of the chapter:

```
> let magnitudesDouble = map (\record ->
    fromSql $  head record :: Double)
    magnitudes
> average magnitudesDouble
1.6950637393767727
```

Summary

This chapter looked at Haskell data types from the perspective of data analysis. You are responsible for the conversions of data within Haskell. This is both a blessing and a curse. Haskell will never manipulate your data without your consent, even in those moments where some data manipulation can be forgiven. We also explored the task of navigating through three common data sources—the Haskell command line where you enter data yourself, csv files, and SQLite3 files. The command line is limited since we can only type so much data ourselves without getting tired. CSV files are the most common source of datasets found on the web. We also explored the difficulties in working with csv files; everything in a csv file is a `String` type and you, as an analyst, have to be dependent on the metadata that accompanies a dataset. We also explored SQLite3, which allows us to leverage two powerful worlds into an environment—functional programming and SQL. Using SQL will give us wonderful versatility over csv files, as we will demonstrate in forthcoming future chapters.

The next chapter will look at the necessary task of cleaning and organizing our datasets. We will look at Haskell's regular expression library in order to filter data based on specific properties of fields.

3
Cleaning Our Datasets

Data originates from various sources (empirical research, historical research, or record keeping). At some point, a human has to consolidate data in a dataset. Humans are creatures who are far from perfect, and this human process of consolidating data will result in tiny imperfections in our datasets. This chapter looks at the techniques that we can use to identify problems in a dataset.

In this chapter, we will cover the following topics:

- Structured versus unstructured datasets
- Creating your own structured data
- Counting the number of fields in each record
- Filtering data using regular expressions
- Searching fields based on a regular expression

Structured versus unstructured datasets

In the last chapter, we navigated data from three different sources: direct keyboard entry, csv files, and SQLite3 files. Data can originate from many more sources than just these. We typically classify the format of the data into two types: structured and unstructured data. Structured data consists of raw data with a degree of organization in the layout. Common examples of structured data include relational or hierarchical databases, CSV, XML, JSON, and YAML file formats. Regardless of the format of the data, the data is organized into a pattern that can be understood by the software (that is, our data is *machine readable*) and meets the criteria set forth in a metadata document.

The following sentence is what most would consider as unstructured data:

> *"Nicknamed "The Wizard" for his defensive brilliance, Smith set major league records for career assists (8,375) and double plays (1,590) by a shortstop"*
>
> *-Wikipedia entry for Ozzie Smith*

While there is a healthy debate on what structured data is, I tend to lean toward two requirements—the dataset should be in a machine-readable format and the dataset should meet the criteria set forth in a metadata document. If there is no machine-readable way to consistently pull values from a dataset, the dataset is classified as *unstructured* data. Unstructured data represents all other datasets. Examples of unstructured data include values that are found in text, the words of someone speaking on a recording in an audio file, the characters on a page from a scanned image, identifying a person in a video clip, or even structured data (like a csv file) that happens to be embedded in unstructured data.

How data analysis differs from pattern recognition

We do have strategies to extract text from images and words from audio recordings and pluck values from sentences. Each of these examples uses a field of computer science known as **pattern recognition**, which attempts to automate the process of defining a structure to unstructured data. While there are many successful techniques to solve this problem in various contexts, they also have a margin of error that is built into their success rate. In order to be considered *structured*, there needs to be perfect accuracy (we get it right the first time) and consistency (this happens every time) to the manner in which data is accessed. Data analysis differs from the field of pattern recognition because the structure of the data is assumed to not be the problem that we are trying to solve.

If your primary source of data is structured in your favorite format, error-free, and distilled to just the records needed to work on your desired problem, then someone has already performed the hardest job in data analysis for you—cleaning the dataset. Cleaning data is the least glamorous part of data analysis. Yet, it consumes most of our time. Datasets frequently come with their own quirks. The typical oddities that should be anticipated are missing values, duplicate records, misspelled identifiers, data outliers, and column values that do not seem to have a consistent type. Likewise, we frequently need to merge columns when it makes sense. We occasionally must split a column when multiple pieces of information are being expressed. During times when you are blessed with too much data, you will have to perform the common task of filtering data.

To perform all these tasks, you will need a scrub brush and a willingness to get the job done. Sometimes, the data is so messy that you will understand why an entire field of science is devoted to organizing data.

Creating your own structured data

If the primary source of your data is unstructured or nonexistent, then we will start from the very beginning. Presented here is my personal workflow to create a single structured dataset:

1. Consider a question that you would like to answer through data along with the necessary data needed to answer that question.
2. Create a metadata document of your desired dataset columns and types.
3. Gather the data related to the problem in one or more unstructured datasets.
4. Convert each unstructured dataset into a machine-readable format.
5. Seek inconsistencies in each dataset and fix them.
6. Align types in each record to match the type defined by your metadata document.
7. Filter columns in each dataset to only the columns defined by your metadata document.
8. Merge your datasets into a single dataset.
9. Identify duplicate records and consolidate them.

If this seems like grunt work, you are correct. Haskell is here to help.

Counting the number of fields in each record

To demonstrate how to find problems in data, we will examine the common problems found in csv files. The first common issue is that the number of fields in each record does not always match the number of columns in the heading line. For this example, I created a simple csv file with an inconsistent number of fields named `poorFieldCounts.csv`. When typing up this file, make sure that the last line ends with a newline character. Some implementations of the Haskell CSV library require the following:

```
Name,FavoriteColor,FavoriteFood
Fred,Orange,Ribs
Wilma,White
Barney,Brown,Pie,Bowling
Betty,Blue,Cake
```

We can write a simple check for this using the following function. This function assumes that `Text.CSV` and `Data.List` have been imported:

```
countFieldsInEachRecord :: CSV -> [Integer]
countFieldsInEachRecord csv = map genericLength (init csv)
```

The preceding function will take an already opened csv file and count the number of fields in each record.

Testing the following statements that shows the number of fields in the heading row and every subsequent record:

```
> let csv = parseCSVFromFile "poorFieldCounts.csv"
> either Left
    (\csv -> Right $ countFieldsInEachRecord csv) csv
Right [3,3,2,4,3]
```

You may see that the function reports the correct information, but this will not be useful if you have several thousands of lines in a csv file. We should filter out records that have exactly one field per column heading so that we can quickly identify the problem areas.

Our next function does just that. The `lineNumbersWithIncorrectCount` function will return a list of the `Integer` pairs. Within each pair, the first integer will represent the line number of a file and the second will report the number of fields that exist in that row:

```
lineNumbersWithIncorrectCount :: CSV -> [(Integer, Integer)]
lineNumbersWithIncorrectCount (fields:csv) = filter
    (\(_, thisCount) -> thisCount /= nfields)
    lineNoCountPairs
  where
    nfields = genericLength fields
    count = countFieldsInEachRecord csv
    lineNoCountPairs = zip [1..] count
```

Executing the preceding code reveals where we should focus our corrections:

```
> either Left
  (\csv -> Right $ lineNumbersWithIncorrectCount csv) csv
Right [(3,2),(4,4)]
```

From the result of the previous code we can see that line 3 only has two fields `(3,2)`, whereas line 4 has four fields `(4,4)`. How you chose to fix your file will be based on the circumstances of your problem, for example, your situation may allow you to ignore these records by either skipping the records or deleting them (always back up your data first).

We will test our function on a csv file that we already know to be correct, such as the earthquake data file used in the previous chapter:

```
> input <- readFile "all_week.csv"
> csv <- parseCSVFromFile "all_week.csv"
> either Left
  (\csv -> Right $ lineNumbersWithIncorrectCount csv)
  csv
Right []
```

Good! An empty list (which is what we expected) is returned.

Our next venture into the cleaning of data depends on whether each record in our csv files has the correct number of fields.

Filtering data using regular expressions

It has been said before that if you have a problem and your solution is to use regular expressions, you now have two problems. **Regular expression** is a term used to represent the language to identify patterns found in text. The language itself is terse (a single character in this language can have a complex meaning). In the open source community, the most identifiable example of regular expressions in use is with the command-line tool, `grep`. The name is derived from an older text editor called `ed`, which had a command called `g/re/p`. Using `grep`, we can search for a pattern of text in a file and filter out anything that does not contain this pattern, for example, let's assume that we have a text file that represents the entire book of the Mark Twain classic, *The Adventures of Huckleberry Finn*. You can download this entire book from Project Gutenberg. I have renamed my text file `huckfinn.txt`.

To identify each line in the file that references the character of Jim in the story, we will use `grep`. Here, I'll use `grep` from the Linux command line, as follows:

```
$ grep Jim huckfinn.txt
```

In this example, `Jim` represents a regular expression. We are looking for any instance in the file where `J` is followed by `i`, which is then followed by `m`.

Creating a simplified version of grep in Haskell

Since this is a book that focuses on Haskell, we will recreate the `grep` tool in Haskell. To do this, we need to make sure that the regular expression library is installed on our system. We can do this using the cabal tool, as follows:

```
cabal install regex-posix
```

The primary usage of `grep` is that the first argument after the name of the command is a regular expression, and all the subsequent arguments represent filenames that need to be searched. We will divide the task of our program into two parts—managing the input arguments and searching files based on a pattern. First, we will search for files based on a pattern. We will begin the file with the following necessary `import` statements:

```
import Text.Regex.Posix ((=~))
import System.Environment (getArgs)
```

We will define our function to search lines in a file based on a regular expression. In the spirit of the original `grep` tool, this function will print lines rather than return matched lines. This can be seen using the following function:

```
myGrep :: String -> String -> IO ()
myGrep myRegex filename = do
    fileSlurp <- readFile filename
    mapM_ putStrLn $
        filter (=~ myRegex) (lines fileSlurp)
```

This function should be relatively straightforward. The `filter` function is the standard tool that is used to filter a list of values based on a Boolean expression. Lines break a slurped file into individual lines. The `=~` operator is from the `Text.Regex` module that allows us to compare each line in a file to a pattern.

Now, we will set up the call to this function, as follows:

```
main :: IO ()
main = do
    (myRegex:filenames) <- getArgs
    mapM_ (\filename -> myGrep myRegex filename) filenames
```

The first argument obtained from `getArgs` (found in the `System.Environment` module) is the regular expression, and the remainder of the list should be our filenames. We will call `myGrep` on each filename with the regular expression and get our results, as follows in the file:

```
$ runhaskell hgrep.hs Jim huckfinn.txt
CHAPTER II. The Boys Escape Jim. Torn Sawyer's Gang. Deep-laid Plans.
Island. Finding Jim. Jim's Escape. Signs. Balum.
CHAPTER XXIII. Sold. Royal Comparisons. Jim Gets Home-sick.
CHAPTER XXIV. Jim in Royal Robes. They Take a Passenger. Getting
CHAPTER XXXI. Ominous Plans. News from Jim. Old Recollections. A Sheep
(...Remaining lines clipped...)
```

Using simple functional programming tools such as `filter` and `map`, we can easily create a command-line tool to parse a number of files based on a regular expression in just a few lines of code.

Exhibit A – a horrible customer database

We need to seek out missing values in our datasets. In the next few examples, we are going to use a simple csv file. The data presented below was randomly generated thanks to the `www.fakenamegenerator.com` website. I have modified several fields to make this dataset purposely bad. The original csv file came with an odd Unicode character embedded as the first character, thus illustrating that even seemingly good csv files can still require cleaning. Here is the file that I named `poordata.csv`:

```
Number,Gender,GivenName,Surname,City,State,Birthday
1,female,Sue,Roberson,Monroe,LA,12/31/1791
2,,George,Chavez,Chicago,IL,11/11/1948
3,male,Dexter,Grubb,Plattsburgh,NY,6/4/1984
4,male,    ,Knight,Miami,Florida,6-21-1951
5,male,Jonathan,Thomas,Fort Wayne,IN,1/15/1967
6,MALE,Brandon,    ,pittsburgh,pa,8/3/1981
7,male,Daniel,Puga,Evansville,,8/19/1988
8,Female,Geneva,Espinoza,Springfield,MA,1992-08-11
9,female,Miriam,Levron,Hicksville,N.Y.,9/7/1965
10,F,Helen,Pitts,Gibsonia,PA,"March 12, 1989"
```

Imagine that you were given this file that represents your company's customer database. Thanks to the infinite wisdom of the developers who designed the registration system, the customers were allowed to freely type their birthday and gender in the birthday and gender fields. The system was not concerned with missing fields and there are a few blank fields in this dataset. If you inspect the `State` column, you will see that some people typed the two-letter capital initials for their state, some typed the initials with periods (see `N.Y.`), and some typed the entire state name (see `Florida`). Whoever was in charge of maintaining this data failed to do an adequate job.

The first thing that you should do is save your originals using the version control software. Overcome the temptation to immediately start fixing the flaws.

Searching fields based on a regular expression

We are going to take what we learned in the previous section and expand it to csv files. We wish to identify every field in the previous csv file that matches a regular expression. These functions will require the `Text.CSV` module (installed in the last chapter) and the `getColumnInCSV` function that we wrote in the `LearningDataAnalyis02` module.

In *Chapter 2*, *Getting Our Feet Wet*, we listed several assumptions that we would be making about csv files. We will now add a new assumption—csv files will have a unique identifier column somewhere in the file. A **unique identifier** column represents a column of identifiers (none duplicated) that represent the data for that row. In a relational database, this would be the primary key field. We will make a second assumption about our data—the column of the unique identifier will be free from errors. Typically, this column is found in the first column of the csv file, but this is not always the case. In the earthquake csv file, the unique identifier column was the 12th out of 15 columns.

Here's a personal story. When I was younger, I created a MySQL database table with a primary key using the `SMALLINT` type for a small business. In the third year of the business, the program failed because the business finally grew to the 65,537th business transaction, which is one more than is allowed by `SMALLINT`. Even the primary key field has the potential to fail if you make foolish design choices.

Here are the import statements needed for our code to work:

```
import Text.CSV
import Data.List
import Text.Regex.Posix ((=~))
import LearningDataAnalysis02
```

It is a good programming strategy to think a little, code a little, and then test a little. The first chunk of code to think/implement/test will represent the heart of the program; the function will apply the regular expression to every field in a record. The function should return three elements for every field that matches the regular expression, the column name in which the field was found, the unique identifier for this record, and the text of the matching field. Since this function depends on `genericIndex`, I wish to remind you that this function can fail if the assumption that the unique identifier exists is not met. This can be seen in the following code snippet:

```
identifyMatchingFields ::
    (String -> Bool)
     -> [String]
     -> [String]
     -> Integer
     -> [(String, String, String)]
  identifyMatchingFields
    myStringCmpFunc record headings idColumnIndex =
      filter
        (\(_, _, field) -> myStringCmpFunc field)
      keyvalue
    where
      nfields = length headings
      keyvalue = zip3
                 (replicate
                   nfields
                   (genericIndex record idColumnIndex)
                 )
                 headings
    record
```

What this function mostly does is juggle each of the necessary input parameters. This function requires a Boolean function used for string comparisons called `myStringCmpFunc`, a record from a csv file called `record`, the headings for each file called `headings`, and the index position of the unique identifier. In the `where` clause of the function, note that a combined list of every unique identifier, column heading, and field is being made using the `zip3` function. This list is filtered just on the field, but when a field does match, the identifier and heading are returned with it.

In our last example, we used the `=~` regular expression comparison operator to facilitate the filtering of the lines of a text file. In this example, we will let you decide as to whether you wish to use simple string comparison functions to identify fields or the far more complex regular expression engine. If you are already comfortable with regular expressions, then we are going to continue using them. If you need a little more time to get used to regular expressions, I hope that you find the guide to regular expressions in Haskell in the appendix of this book helpful and then return to this chapter. Either way, the function only needs to be written once.

We will conduct a test with a small sample. A regular expression of `Journ` will match anything that includes this sequence of characters, which is demonstrated in the following code:

```
> identifyMatchingFields (\x -> x =~ "Journ") ["1", "Clark Kent",
"Journalist", "Metropolis"] ["Id", "Name", "Profession", "Location"] 0
[("1","Profession","Journalist")]
```

A regular expression of `Hero` should not match anything, since this sequence does not exist in this data:

```
> identifyMatchingFields (\x -> x =~ "Hero") ["1", "Clark Kent",
"Journalist", "Metropolis"] ["Id", "Name", "Profession", "Location"] 0
[]
```

Of course, we do not have to use a regular expression with our newly crafted function. If you need to search for something simple, the use of regular expressions is going to be overkill. You can search for `Metropolis` in your data if you are looking for fields that exactly contain the word, Metropolis. The predicate function in the `identifyMatchingFields` function allows us to be versatile in the methodology of how fields are searched. You can use a simple comparison, a regular expression, or perhaps a method that I did not consider:

```
> identifyMatchingFields (== "Metropolis") ["1", "Clark Kent",
"Journalist", "Metropolis"] ["Id", "Name", "Profession", "Location"] 0
[("1","Location","Metropolis")]
```

Once you feel satisfied with your testing, you can move on to building a function that matches every field in a csv file. This function will take a string comparison function, an open csv file, and a string representing a column heading and return every field that matches the given comparison function along with the field's unique identifier and heading, as follows:

```
identifyInCSV ::
  (String -> Bool) -> CSV -> String ->
    Either String [(String, String, String)]
    identifyInCSV myFieldFunc csv idColumn =
      either
      Left
      (\ci -> Right $ concatMap
        (\record ->
        identifyMatchingFields
        myFieldFunc record (head csv) ci
        )
        (tail csv)
      )
      columnIndex
    where
      headings = head csv
      columnIndex = getColumnInCSV csv idColumn
```

By now, the preceding code should be familiar to you. We used the `getColumnInCSV` function (written in the last chapter) to get the index of a column heading. Since this function call has the capacity to return an error, we must wrap the returned expression in an `either` clause. The `concatMap` function does most of the work in the function by calling the `identifyMatchingFields` function repeatedly and concatenating all the returned lists into one.

First, we will test the list with a regular expression that we know will return a value. Only one record uses the state abbreviation `PA`:

```
> csv <- parseCSVFromFile "poordata.csv"
> either (\error -> Left "CSV Problem") (\csv -> identifyInCSV (\x -> x
=~ "PA") csv "Number") csv
Right [("10","State","PA")]
```

Next, we will test the list with a regular expression that should match multiple fields. Several records use the word male to represent a gender. Note that this also returns everything that matches `female`. Regular expressions do not limit themselves to matching complete strings:

```
> either (\error -> Left "") (\csv -> identifyInCSV (\x -> x =~ "male")
csv "Number") csv
Right [("1","Gender","female"),("3","Gender","male"),("4","Gender","male"
),("5","Gender","male"),("7","Gender","male"),("8","Gender","Female"),("9
","Gender","female")]
```

Again, after we are satisfied with our testing, we can craft a primary function, which will be used to call our helper functions:

```
identifyInCSVFile ::
  (String -> Bool) ->
    String ->
    String ->
    IO (Either String [(String, String, String)])
  identifyInCSVFile myStringCmpFunc inFileName idColumn = do

  records <- parseCSVFromFile inFileName
  return $ either
    (\err ->
    Left "This does not appear to be a CSV file")
    (\csv ->
      identifyInCSV myStringCmpFunc (init csv) idColumn
    )
    records
```

Locating empty fields in a csv file based on a regular expression

Using our newly crafted function, we will attempt to locate each field in our csv file which is empty or nearly empty. Because a regular expression can have its match anywhere within a field, we must force our regular expression to start at the beginning and end at the end of a field. To ensure that a regular expression starts at the beginning, we will begin that expression with the `^` symbol. To ensure that an expression ends at the end of a string, we will end the expression with `$`. Thus, the regular expression `^$` represents an empty field:

```
> identifyInCSVFile (\x -> x =~ "^$") "poordata.csv" "Number"
Right [("2","Gender",""),("7","State","")]
```

The second record contains no data for the `Gender` column and the seventh record contains no data for the `State` column.

How do we identify nearly empty data? In this case, we have data, but it is not useful because it has some space characters. We wish to find any field with 0 or more whitespace characters. In order to catch all the forms of whitespace characters (including space, tabs, and new lines), we will use the special shortcut atom called `\\s` (for space) in the following command, and it needs to be modified with the `*` modifier to represent 0 or more occurrences of that atom. We also need to add the `^` and `$` positional modifiers. Our final whitespace expression is `^\\s*$`. This regular expression translates as *At the start of a string, there should exist 0 or more whitespace characters, followed by the end of the string*.

```
> identifyInCSVFile (\x -> x =~ "^\\s*$") "poordata.csv" "Number"
Right [("2","Gender",""),("4","GivenName","    "),("6","Surname","
"),("7","State","")]
```

Since our dataset is a work of fiction, we can fill in the gaps with fictional data. When working with gaps in your dataset, you should probably seek the advice of a domain expert. At the very least, filter out lines containing gaps (back up your data first) until you can fill in those gaps with accurate information.

Crafting a regular expression to match dates

There are still many problems with our csv file. Let's focus on the `date` column. We wish to craft a function that will be able to pull just the date of births from our dataset in the `Birthday` column. The trick to solve this lies in writing a regular expression that matches everything except for the accurate-looking dates and then writing a short filter to exclude everything except for our desired column. We will achieve this by building on our last function.

Without going into detail, an acceptable regular expression to identify a date in the form of `MM/DD/YYYY` (the American style of writing dates) will be `^[1-9][0-9]?/[1-9][0-9]?/[12][0-9][0-9][0-9]$`. By applying the `=~` regular expression comparison operator to our data with this expression, we will only return the correctly formatted dates in our file (which is useless at this stage). We wish to return the poorly-formatted dates. To get everything that does not appear to be a date, we will wrap the evaluation expression with Haskell's `not` operator. Then, we will filter the data that does not match our desired column.

Here is the function that is used to take the data returned by `identifyInCSVFile` and filter it based on a column name:

```
identifyInCSVFileFromColumn ::
  (String -> Bool) -> String -> String -> String ->
    IO (Either String [(String, String, String)])
  identifyInCSVFileFromColumn
    myRegexFunc inFileName idColumn desiredHeading = do
      allFields <- identifyInCSVFile
        myRegexFunc inFileName idColumn
        return $ either
        Left
        (\af -> Right $
          filter
            (\(_, heading, _) ->
            heading == desiredHeading
            )
         af
         )
       allFields
```

By allowing `identifyInCSVFile` to perform our heavy lifting, we can reuse our old code in the following way:

```
> identifyInCSVFileFromColumn  (\x -> not (x =~ "^[1-9][0-9]?/[1-9][0-
9]?/[12][0-9][0-9][0-9]$")) "poordata.csv" "Number" "Birthday"
Right [("4","Birthday","6-21-1951"),("8","Birthday","1992-08-
11"),("10","Birthday","March 12, 1989")]
```

Good. By identifying just the poorly-formatted dates from the `Birthday` column, we can correct them. The majority of the dates are in the `MM/DD/YYYY` format. So, we will fix these data values to match the majority format. I edited the file so that the incorrect birthdays are now `06-21-1951`, `08-11-1992`, and `03-12-1989`. After manually correcting the data (make backups first) and naming this file `poordataFixed.csv`, we will test it again, as follows:

```
> identifyInCSVFileFromColumn  (\x -> not (x =~ "^[1-9][0-9]?/[1-9][0-
9]?/[12][0-9][0-9][0-9]$")) "poordataFixed.csv" "Number" "Birthday"
Right []
```

An empty list tells us that all the fields matched the regular expression in the `Birthday` column.

Summary

Cleaning is not only the most important but also the least glamorous phase of data analysis. With Haskell and the power of regular expressions, we can quickly identify areas with large quantities of data that need our attention. We left our cleaning problem incomplete in this chapter. There is still plenty of data left to clean. The `Gender` and `State` columns need some serious work. They are left as an exercise for you to learn how to craft regular expressions to quickly identify the fields that require your attention.

We also discussed the unclear border between what is meant by the terms, *structured data* and *unstructured data*. I applied two pieces of criteria for structured data—the data is in a machine-readable format and the data adheres to a metadata document standard. Our example dataset is still a long way from being *structured*. We assume that the person who aggregated this data had a metadata document in mind, but that didn't stop us from performing a lot of cleaning.

Our next chapter is going to put cleaning aside. We will explore data visually, allowing the data to speak for itself. It is also the key technique that is used to develop some assumptions about our data. In data analysis, the plotting of data is a method of speculation, and you will see that through this speculation, you can allow ideas to flourish. However, it will be through the subsequent chapters that we will learn how to check whether our speculations are correct.

4
Plotting

In this chapter, we will explore data with visualization. Pictures help us tell stories better than words can. Like words, pictures can be used to lead or mislead the reader's thinking. The Yahoo! Finance website has a history of each publicly traded stock in the American Stock Exchange in the csv file format. We will demonstrate how to convert a csv file representing a company's closing price history and visualize that history. We will then demonstrate how to compare multiple companies' share prices on the same plot.

In this chapter, we cover the following:

- Introducing the Haskell library EasyPlot to plot data
- Simplifying access to data in SQLite3
- Plotting data from a SQLite3 database
- Plotting a subset of a dataset
- Plotting data passed through a function
- Plotting comparisons of multiple datasets
- Plotting a moving average
- Plotting a scatterplot

Plotting data with EasyPlot

Data visualization is the craft of using art to assist the reader in answering questions about data. While it is possible to answer the same questions using only words, pictures will bring data to life in a manner that words cannot. We wish to visualize our datasets using Haskell and SQLite3. To do this, we are going to use the open source tool gnuplot (a popular graphing tool for building visualizations of academic data) and the Haskell interface to gnuplot called EasyPlot. EasyPlot is among the easiest plotting tools to learn in Haskell, with the disadvantage that it is limited in its feature set.

Visualizing data, much like writing about data, requires some criteria for establishing what is (and is not) a good visualization of data. Here are the simple rules that I follow to make my own data visualizations:

- **Does this visualization contribute to the understanding of a dataset?** Art is a powerful tool, and, to quote a famous American comic book, "with great power comes great responsibility." It is the job of the analyst to put the data into a context that does not mislead the reader into a false sense of understanding. Some examples of misleading the reader would be cropping out data points that contradict the writer's interpretation of the data, using colors in a manner that allow elements to be confused, or trying to express too much information in a single chart. Some of these design choices might be seen as unethical in certain contexts.
- **Does the visualization help to answer one or more questions?** All visualizations should make an attempt at answering questions that the reader might have regarding data. Visualizations give us an opportunity to express variables in a concise manner, which allows us to be expressive about the complexities of the data without having to be burdensome in those same details. The reader should gain at least one insight into the data that the creator of the work did not expect. The goal of a good data visualization is to allow the reader to explore rather than to hamstring the reader into a narrow context.
- **Can the visualization be simplified and still answer the reader's questions?** The scientist Carl Sagan wrote about using Occam's Razor in the quest to explain data- *this convenient rule-of-thumb urges us, when faced with two hypotheses that explain the data equally well, to choose the simpler*. The same could be said of data visualizations; when faced with two visualizations that explain data equally well, choose the simpler visualization. If a visualization is too crowded with information, you are encouraged to split each idea represented into its own visualization.

In this chapter, we will explore the `Graphics.EasyPlot` package, but to do that we must first install gnuplot and the EasyPlot library. Using the `apt-get` command in Debian-based Linux distributions, you can download gnuplot using the following:

```
sudo apt-get install gnuplot
```

You can also install the `Graphics.EasyPlot` package in Haskell using the `cabal` command:

```
cabal install EasyPlot
```

Simplifying access to data in SQLite3

In *Chapter 2*, *Getting Our Feet Wet*, we explored how to access data in our newly created SQLite3 databases. Data in SQLite3 can be expressed in terms of the `REAL`, `INTEGER`, and `TEXT` data types, but our Haskell library for retrieving data returns data of the `SqlValue` type. There isn't a convenient way to ask Haskell to give us back our data in the original format, so we will come up with one ourselves.

Let's begin by creating our library for this chapter, `LearningDataAnalysis04.hs`. We will be creating several functions to ease the difficulty in interacting with SQLite3 databases. This new file will begin with the following statements. Note that this file will require the `LearningDataAnalysis02.hs` file created in *Chapter 2*, *Getting Our Feet Wet*:

```
module LearningDataAnalysis04 where
import Data.List
import Database.HDBC.Sqlite3
import Database.HDBC
import Graphics.EasyPlot
import LearningDataAnalysis02
```

The three basic data types used by SQLite3 are `REAL`, `INTEGER`, and `TEXT`. The Haskell equivalents of these types are double, integer, and string. We need a function that accepts a SQLite3 result and a column number and converts that data column to a particular data type. Because of these assumptions, we also assume that you have taken the time to inspect your datasets for irregularities and corrected them. The information on how to quickly find and clean your data can be found in *Chapter 3*, *Cleaning Our Datasets*. The function introduced here will do a sufficient job of pulling information from a SQLite3 database:

```
readIntegerColumn :: [[SqlValue]] -> Integer-> [Integer]
readIntegerColumn sqlResult index = map (\row -> fromSql $
  genericIndex row index :: Integer) sqlResult

readDoubleColumn :: [[SqlValue]] -> Integer -> [Double]
readDoubleColumn sqlResult index = map (\row -> fromSql $
  genericIndex row index :: Double) sqlResult

readStringColumn :: [[SqlValue]] -> Integer -> [String]
readStringColumn sqlResult index = map (\row -> fromSql $
  genericIndex row index :: String) sqlResult
```

In preparing this chapter, I found it convenient to craft a function for the purpose of querying a not-yet-open database file. This simple function will open a connection to a SQLite3 database, send a query and get the results, close the connection, and return the results:

```
queryDatabase :: FilePath -> String -> IO [[SqlValue]]
queryDatabase databaseFile sqlQuery = do
  conn <- connectSqlite3 databaseFile
  let result = quickQuery' conn sqlQuery []
  disconnect conn
  result
```

We are all set to begin our discussion about using the EasyPlot library to plot data.

Plotting data from a SQLite3 database

In order to plot data, we need data to plot! For this, we will be using data representing the stock market of the United States, found on the Yahoo! Finance website. Yahoo! Finance has a wealth of information on every publicly traded company currently being traded in the United States. Conveniently, Yahoo! allows users to download the entire closing share value history of these companies into a csv file format, free of charge.

Let's begin by downloading the entire history of the Apple company from Yahoo! Finance (`http://finance.yahoo.com`). You can find Apple's content by performing a *quote look up* from the Yahoo! Finance homepage for the symbol **AAPL** (that's two *As*, not two *Ps*). From this page, find the link for **Historical Prices**. On the **Historical Prices** page, identify the link that says **Download to Spreadsheet**. The full link to Apple's historical prices can be found here:

```
http://real-chart.finance.yahoo.com/table.csv?s=AAPL&d=10&e=10&f=2014
&g=d&a=11&b=12&c=1980&ignore=.csv
```

We should take a moment to explore our dataset. Here are the column headers in the csv file:

- `Date`: A string representing the date of a particular day in Apple's history
- `Open`: The opening value of 1 share
- `High`: The high trade value over the course of this day
- `Low`: The low trade value over the course of this day
- `Close`: The final price of the share at the end of this trading day
- `Volume`: The total number of shares traded on this day
- `Adj Close`: A variation on the closing price that adjusts for dividend payouts and company splits

One other feature of this dataset is that each of the rows are written into the table in a chronologically reverse order. The most recent date in the table is first. The oldest is last.

Yahoo! Finance provides this table under the unhelpful name `table.csv`. I renamed the csv file provided by Yahoo! Finance to `aapl.csv`. Within the Haskell command line, convert this file to a SQLite3 database using the `convertCSVFileToSQL` function written in *Chapter 2, Getting Our Feet Wet*.

```
> :l LearningDataAnalysis02.hs
> convertCSVFileToSQL "aapl.csv" "aapl.sql" "aapl" ["date STRING", "open
REAL", "high REAL", "low REAL", "close REAL", "volume REAL", "adjclose
REAL"]
Successful
```

For our purposes, the most important column in this table is the `Adj Close` column, which gives us the clearest depiction of the data. We introduce a new function called `pullStockClosingPrices`, which pulls data from a database. This function will require a database file and the name of a database containing the financial data of a stock based on the format provided by Yahoo! Finance. This function will grab each record's `rowid` (which SQLite3 gives all tables automatically) and the `adjclose` field.

Note that we apply the `reverse` function to the `rowid` column; this is because we want the data values to be represented in chronological order:

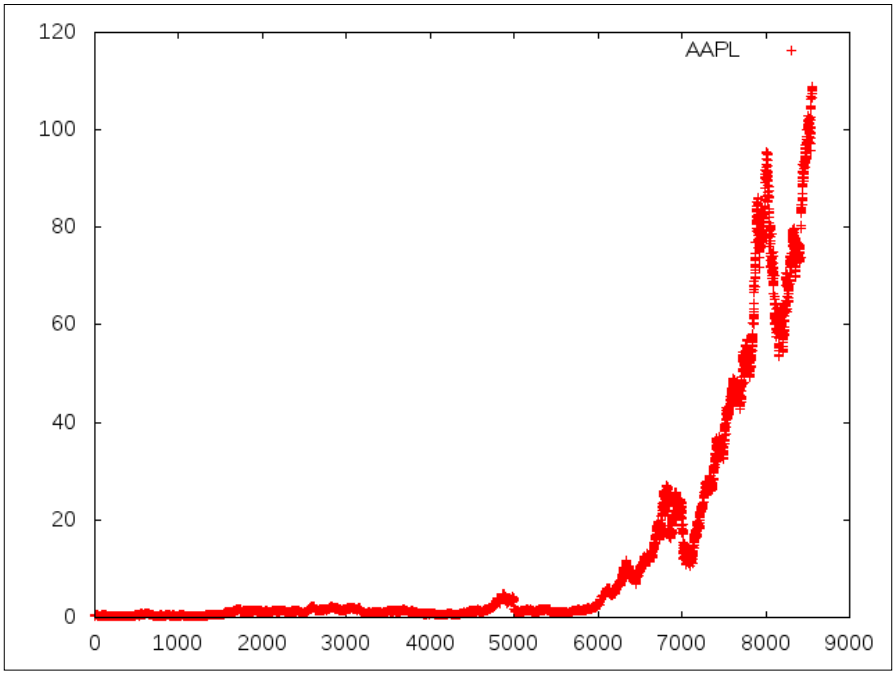
```
pullStockClosingPrices :: String -> String -> IO [(Double,
  Double)]
pullStockClosingPrices databaseFile database = do
  sqlResult <- queryDatabase
    databaseFile ("SELECT rowid, adjclose FROM " ++ database)
    return $ zip
      (reverse $ readDoubleColumn sqlResult 0)
      (readDoubleColumn sqlResult 1)
```

Great. We have put together enough pieces of the puzzle to create our first visualization. We will be plotting the entire history of Apple's share price. Each `rowid` represents an x-coordinate in a scatterplot, and each `adjclose` represents the y-coordinate. We are going to pass a list of doubles defined as `[(Double, Double)]` to our plot function, where each tuple represents an (x, y) pair.

```
> :l LearningDataAnalysis04
> aapl <- pullStockClosingPrices "aapl.sql" "aapl"
> plot (PNG "aapl.png") $ Data2D [Title "AAPL"] [] $ aapl
True
```

Look in your current working directory and you should find a file titled `aapl.png`. Open it up.

The following screenshot is a typical default chart created by EasyPlot. We see the entire history of Apple's stock price. For most of that history, Apple's adjusted share price was less than $10 per share. At about the 6,000th trading day, we see the quick ascension of the share price to over $100 per share.



Exploring the EasyPlot library

EasyPlot is a wrapper for the gnuplot software and provides a simple interface to create basic plots. We have limited access to the full power of gnuplot using this library, but that's fine. We can create outstanding plots with just this library.

We do not use the more advanced functionality of EasyPlot that does allow us to create plots based on direct gnuplot commands.

Here is the type signature to the plot function. There's only two arguments to the function:

```
plot :: TerminalType -> a -> IO Bool
```

A complete documentation of this function can be found on the Hackage page for EasyPlot. We believe that it's beneficial for the reader to cover this library in more detail by navigating here:

`http://hackage.haskell.org/package/easyplot-1.0/docs/Graphics-EasyPlot.html`

The `TerminalType` argument is how we specify the output of the graph. The output allows us to tell EasyPlot that we wish to view this plot as a file or as an interactive plot. In our example, we specified that we wanted a PNG image followed by the image name. We could have easily specified JPEG or GIF (for images), **LaTeX** (for academic publications), PDF (for Adobe PDF documents), or PS (for encapsulated postscript documents).

Each of these arguments requires an output file name similar to the PNG call used in our example. There are also specific interactive plots that can be used for your respective operating system; Aqua is for Mac systems, Windows is for Windows systems, and X11 is for Linux systems with a graphical user interface installed. Each of these arguments can be used by themselves.

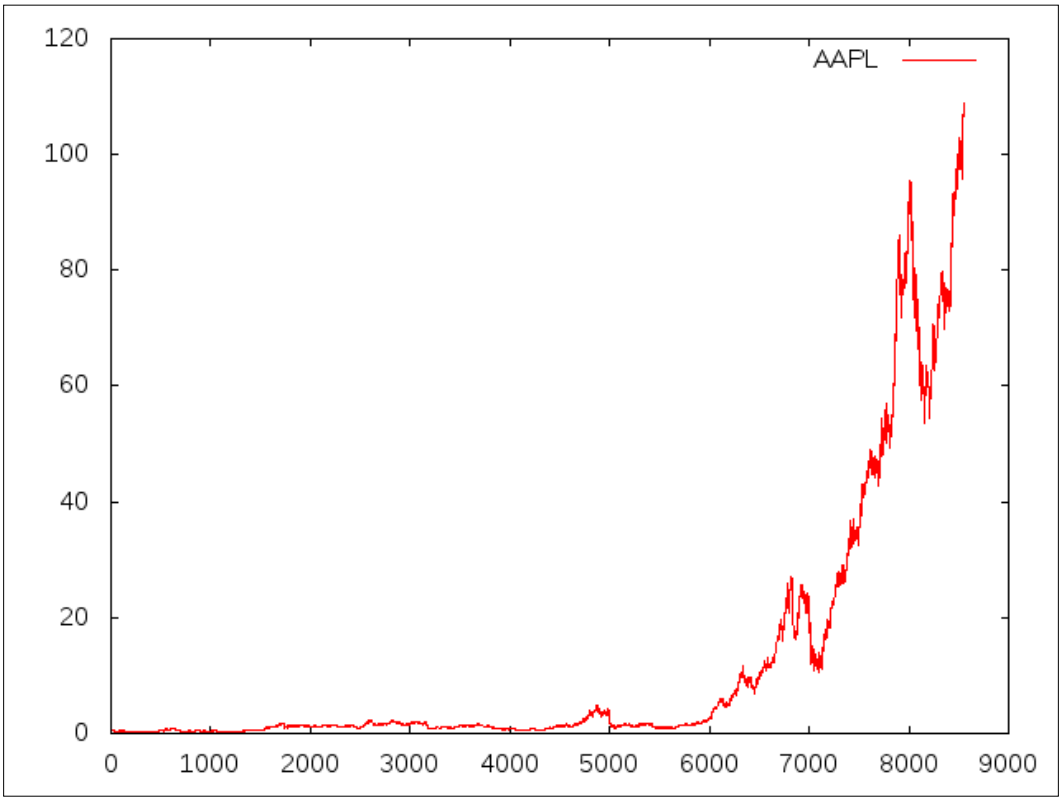
The second argument, `a`, is how we specify one or more datasets for plotting. The argument `a` can be a list of datasets, a single dataset, or even a gnuplot command, but we will commonly supply each dataset wrapped in either the **Graph2D** type or the **Graph3D** type.

The Graph2D type can be constructed using three different types of two dimensional plots: Data2D (for **point cloud datasets** that we use in this chapter), Function2D (for plotting based on a mathematical Haskell function), and Gnuplot2D (for plotting based on a mathematical function in a string that will be passed directly to gnuplot). The Graph3D type has each of these respective constructors, with the obvious difference that they require three-dimensional data and functions and are named Data3D, Function3D, and Gnuplot3D.

The default style of plotting two-dimensional point cloud data is to use the cross marker for each data point. With 9,000 points, the crosses overlap and our dataset is easily confused. We should express our data using a line plot rather than with crosses:

```
> plot (PNG "aapl_line.png") $ Data2D [Title "AAPL", Style Lines] [] $
aapl
True
```

Identify the `aapl_line.png` plot in your current working directory. This is a better representation of the data in my opinion:



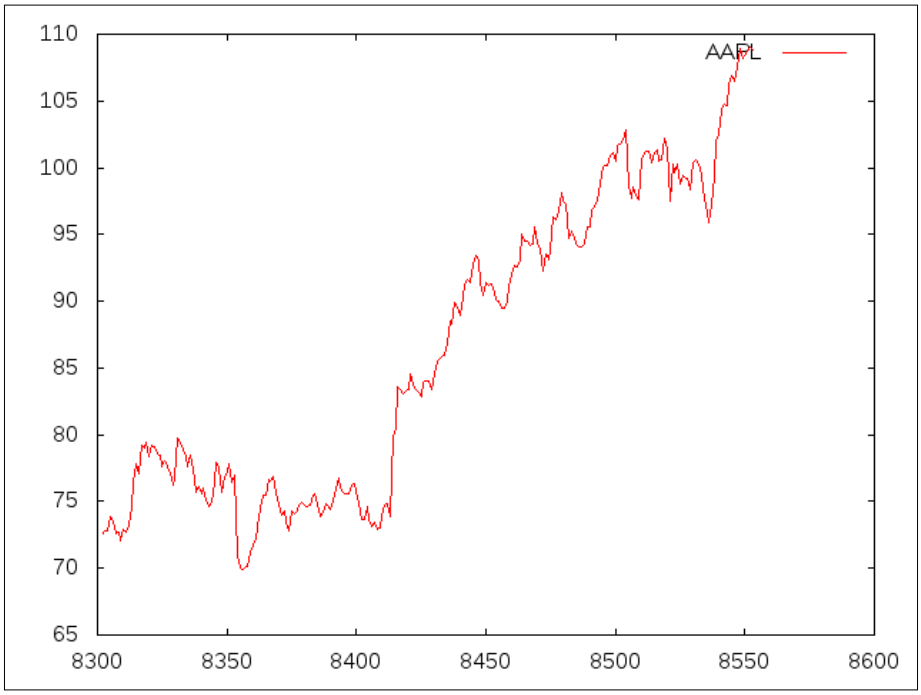
Plotting a subset of a dataset

This dataset is interesting, but we aren't trying to answer any questions with our data. That's fine. Data visualizations can be used to answer questions or it can be used as a tool for exploring data and discovering questions that need to be answered. We can see in the previous chart that the most interesting part of the data is the most recent jaggedness on right side of the graph representing the most recent trading days.

Investors like to look at data through various windows of time, say, the past five days, one month, six months, or one year of trading. Let's look at this data by using Haskell to take the most recent year of trading data. According to Wikipedia, there are typically 252 trading days in a calendar year on the **New York Stock Exchange** (**NYSE**). With this information, we use the `take` function to plot the data of the past year in the following manner:

```
> plot (PNG "aapl_oneyear.png") $ Data2D [Title "AAPL", Style Lines] [] $
take 252 aapl
True
```

This gives the following chart as the output:



By reading this plot we can see that, about a year ago, AAPL was trading for just under $75 per share and has grown in value to over $105 per share. We can also see the fluctuation in the share price, but from this perspective it's too soon to apply any sort of meaning to these fluctuations.

Plotting data passed through a function

Plotting the share price over time is a quick way of glancing at data and seeing if this data has increased or decreased over time. A company's share price increasing over time is usually associated with the attributes of a good company. If we had invested in Apple a year ago, our share value would have increased to just over $30 per share. Is $30 per share investment over the course of a year a good investment or a bad one? The answer to this depends on the original share value. We could answer this question with the previous chart, but it could be made easier. We need to examine this share value through the perspective of the share's percentage of change over time.

To compute the percentage of change, we need two values: the original value and a newer value. If the newer value is greater than the original, we obviously have a positive percent change. If the newer value is lower, we have a negative percent change.

In the following formula, *x1* represents the *first* data value in our dataset. In the context of our analysis of Apple's share value, it's the share price one year ago from the date which we pulled our data. The value of *xi* represents *any* data value in our dataset (including the first value). When every *xi* equals *x1* in our formula, then it evaluates to 0 percent change. Every other value should produce either a increase or a decrease percentage:

$$percent\ change(value, first) = 100.0 \times \frac{value - first}{first}$$

Here, *xi* is associated with the variable value and *x1* is associated with the first variable value. We should represent this formula as a Haskell command:

```
percentChange :: Double -> Double -> Double
percentChange value first = 100.0 * (value - first) / first
```

The `percentChange` function only computes a single percent change at a given point in our data. We need a function that allows us to map this function to all the data points. The `applyPercentChangeToData` function does the trick for us in the following manner:

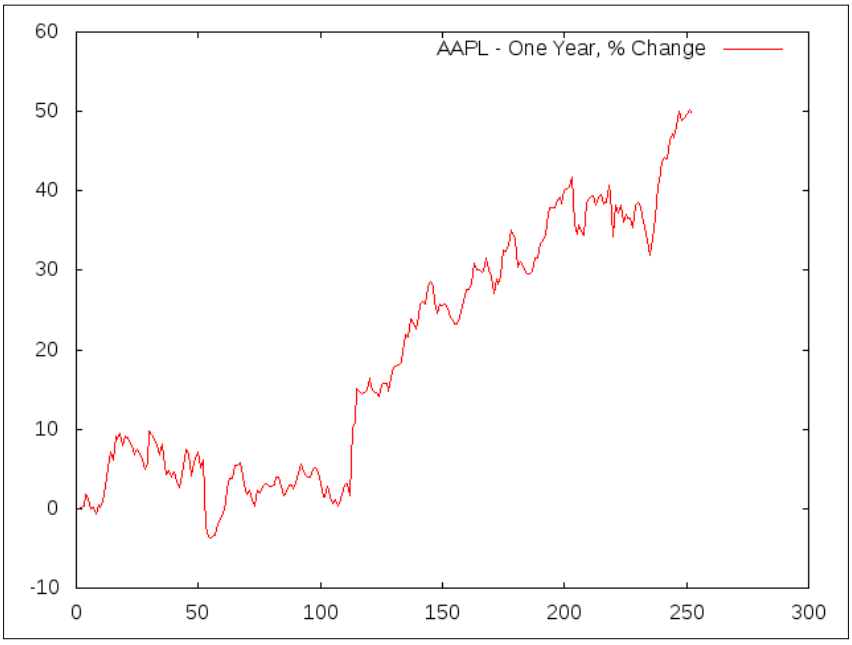
```
applyPercentChangeToData :: [(Double, Double)] -> [(Double,
  Double)]
applyPercentChangeToData dataset = zip indices scaledData
  where
    (_, first) = last dataset
    indices = reverse [1.0..(genericLength dataset)]
```

```
scaledData = map
  (\(_, value) -> percentChange value first)
  dataset
```

This is the function that we will use for plotting the percent change of a share price. This function will also renumber the starting value of each data point to begin with 1. This will be essential for aligning the datasets in the next section. We should call our newly created `percentChange` and `applyPercentChangeToData` functions. We are also using the combination of `snd` and `last` functions to find the first data value in this dataset (remember, this dataset is listed in the chronologically reverse order).

```
> let aapl252 = take 252 aapl
> let aapl252pc = applyPercentChangeToData aapl252
> plot (PNG "aapl_oneyear_pc.png") $ Data2D [Title "AAPL - One Year, %
Change", Style Lines] [] $ aapl252pc
True
```

Open up `aapl_pastyear_pc.png` to find the following new image. You might notice that it's the same line as our previous image. That's because the percent change formula (when applied to a full dataset) is a simple scaling of the data. The structure of the data is perfectly retained in the scaling. We've kept all of the peaks and valleys.



The only aspects of the graph that have changed are the x-axis and the y-axis, which now report the percent change of the data rather than the share price. The first data point in this chart is a 0 percent change, as we should expect. From this chart, we see how much Apple's share price has grown with respect to our original investment. Most of the chart is above the 0 percent change. At the right extremity of the chart, we see what a one-year investment looks like; we would have seen an almost 50 percent return on our original investment.

We need to express a few ideas on the topic of the company share value; this value represents a bit of fiction. Some of the value comes from the tangible assets of the company and some of the value comes from the public perception of that company (which is quantified in the daily buying and selling of shares due to countless subjective events). The share value multiplied by the total number of outstanding shares is called the **market capitalization**, or the total value of a company. The market capitalization is still a bit of fiction, since this value still incorporates the public perception of the company into the overall value.

There are many strategies in stock market investing and your author does not advocate one strategy over another. You should investigate every strategy that comes your way with a healthy dose of skepticism. But we do wish to highlight one strategy just for the ease at which it helps us to convey the ideas behind data visualization: simple growth investing. The growth investment strategy encourages the investor to seek out and buy the shares of companies that are growing faster than the average rate at which the overall market grows and to sell when those companies no longer beat the average growth rate.

For example, if you are trying to decide whether to invest in *Company A* or *Company B*, you should compare the growth rates of these two companies. If *Company A* is growing faster than *Company B*, you should invest in *Company A*. Otherwise, invest in *Company B*. Using this strategy, the investor should consistently beat the market average at the time expense of reevaluating the growth rate of holdings at regular intervals.

Before you race to the bank and take out a loan to throw money at the stock market, you should be aware of the weaknesses to the growth investing strategy. Growth investing ignores the market capitalization of a company (small companies tend to have high growth at a larger risk), the dividend payouts by a company (a value returned by the company that is not reflected in the share price of a stock), the multitude of the popular metrics used by investors to evaluate companies, the public perception of the company, the public perception of the industry, or the public perception of the entire market.

Growth investing only looks at the day-to-day changes in the share price of a company. Growth investing helps investors identify companies that outperform the market but does not guarantee that the current performance will continue in the future for any length of time. You have been warned. If you invest, invest with the understanding that every investment is a risk.

Plotting multiple datasets

This chapter isn't about investing. It's about crafting compelling data visualizations. I picked Apple's stock price for these visualizations because Apple was in a position to tell a compelling story through their share price at the time at which this book was being written. I will make no predictions or promises about the future value of Apple's share price.

We would like to get a better picture of Apple's 50 percent increase in share price over one year by comparing Apple's share price with the share price values of other companies. As we stated earlier, all share prices are a bit of fiction. Better stated, we would like to compare the percent change of multiple companies' share values through a data visualization.

Go back to Yahoo! Finance and download the historical share price csv files for Google (symbol: GOOGL) and Microsoft (symbol: MSFT). I named their respective csv files `googl.csv` and `msft.csv`.

Here's Google's share history:

```
http://real-chart.finance.yahoo.com/table.csv?s=GOOGL&d=10&e=10&f=201
4&g=d&a=11&b=12&c=1980&ignore=.csv
```

And, here's Microsoft's share history:

```
http://real-chart.finance.yahoo.com/table.csv?s=MSFT&d=10&e=10&f=2014
&g=d&a=11&b=12&c=1980&ignore=.csv
```

We are going to perform the same steps that we did with Apple's data and create a plot containing Apple, Google, and Microsoft's percent change in a share price plot.

First, Google:

```
> convertCSVFileToSQL "googl.csv" "googl.sql" "googl" ["date STRING",
"open REAL", "high REAL", "low RDEAL", "close REAL", "volume REAL",
"adjclose REAL"]
Successful
> googl <- pullStockClosingPrices "googl.sql" "googl"
> let googl252 = take 252 googl
> let googl252pc = applyPercentChangeToData googl252
```

Next, Microsoft:

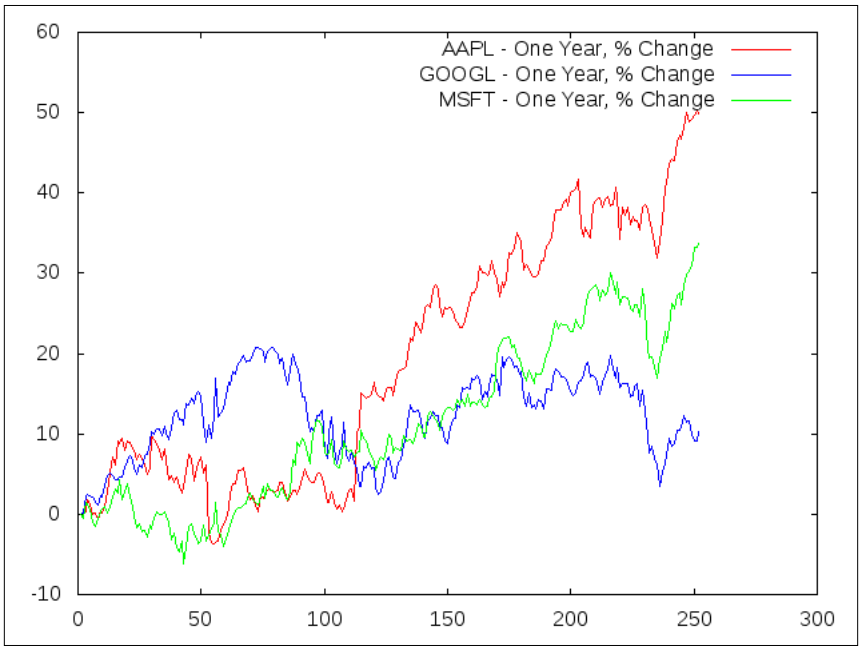
```
> convertCSVFileToSQL "msft.csv" "msft.sql" "msft" ["date STRING", "open
REAL", "high REAL", "low REAL", "close REAL", "volume REAL", "adjclose
REAL"]
Successful
> msft <- pullStockClosingPrices "msft.sql" "msft"
> let msft252 = take 252 msft
> let msft252pc = applyPercentChangeToData msft252
```

Next, we are going to plot these three datasets with a single call to plot:

```
> plot (PNG "aapl_googl_msft_pc.png") [Data2D [Title "AAPL - One Year, %
Change", Style Lines, Color Red] [] aapl252pc, Data2D [Title "GOOGL - One
Year, % Change", Style Lines, Color Blue] [] googl252pc, Data2D [Title
"MSFT - One Year, % Change", Style Lines, Color Green] [] msft252pc]
True
```

It's a long line, but it illustrates how to combine multiple datasets into a single plot. Note that each dataset is represented with its own Data2D constructor, as well as having its own title, color, and plotting style. Every dataset also has an empty list (required) that we have not discussed up to this point. This empty list is used to include preferences on how functions are plotted. We aren't using these options just yet, but we will in future chapters.

Let's explore the chart that we created:



The benefit of using the `percentChange` function on each of our datasets is that the function will automatically scale the data so that we can enjoy an accurate comparison of the change in the datasets over time. From here, we can see that Apple clearly grew the most over the past year, followed by Microsoft at just over 30 percent, followed by Google at close to 10 percent.

We can also identify when during the past year the share price was below 0. Apple and Microsoft both briefly dipped below 0, while Google stayed in positive territory for the entire year. When evaluating these companies with the criteria set forth in a growth investing strategy, it appears that the clear winner among the three is Apple for this one year period.

Plotting a moving average

Look back at each of our charts. We can see that stocks tend to move up and down with jagged shifts. These peaks and valleys represent the day-to-day trade noise. A moving average is a way of looking at the value of a stock without having to look at the "ugly" jaggedness of the true price. If we computed the average of the entire past year's data, it would be a single value and it wouldn't be very useful. If we sectioned off the first, say, 20 values, into a window, and averaged those values, we would be representing the first 20 values as its own data value. We could then shift our window forward one data value and average those 20 values. It wouldn't be much different from the first, but the overall trend would be maintained.

In order to create a moving average, we will be using the `average` function that was created at the beginning of *Chapter 2*, *Getting Our Feet Wet*. You should find that function in the `LearningDataAnalyis02.hs` file. This new function, `movingAverage`, will reside in our `LearningDataAnalyis04.hs` file.

The `movingAverage` function will take a list of `Double` values and a window size (represented by an integer). This method uses recursion to march through the list. This function will compute the average of the first window values of the list. If there are more than just window elements in values, the function will pass the tail of values to itself for more processing.

```
movingAverage :: [Double] -> Integer -> [Double]
movingAverage values window =
 | window >= genericLength values = [ average values ]
 | otherwise = average
               (genericTake window values):(movingAverage (tail
               values) window)
```

Good. 2D plots require that we have an *x*-value and a *y*-value for plotting. We need to assign an *x*-value to each of these *y*-values. The following function will create an *x* pair for every *y* moving average with a starting index that is equal to the window size (call this function with a window size of 20 and the first index will be at position 20):

```
applyMovingAverageToData ::
  [(Double, Double)] -> Integer ->  [(Double, Double)]
applyMovingAverageToData dataset window =
  zip [fromIntegral window..] $ movingAverage
    (map snd (reverse dataset))
    window
```

The word *size* here is a bit misleading, for the list doesn't use infinite memory; consider just *an infinite list* instead.

This command is relatively straightforward except for our use of Haskell's lazy evaluation. Note that we have an infinite list in the command at `[fromIntegral window..]`. We are creating a range of values starting with the value window, but we never told Haskell where to end. The list is infinitely long, but ZIP knows that it needs to stop when the `movingAverage` function stops.

This is Haskell's beautiful (and powerful) trait of being lazy; Haskell will only do the work that is necessary to get the job done, and the job is done when the `movingAverage` function stops.

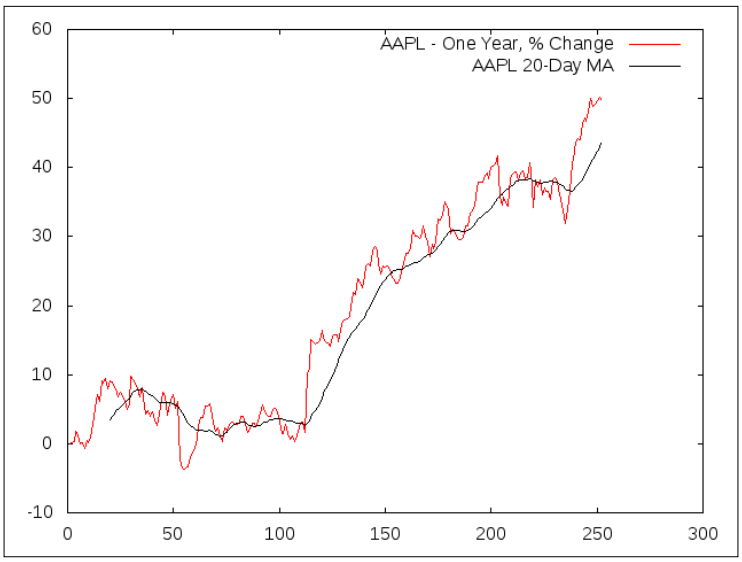
We are going to apply this function to our Apple percent change data from the past year using a 20-day moving average:

```
> aapl <- pullStockClosingPrices "aapl.sql" "aapl"
> let aapl252 = take 252 aapl
> let aapl252pc = applyPercentChangeToData aapl252
> let aapl252ma20 = applyMovingAverageToData aapl252pc 20
```

We will be plotting the data with the color red and the 20-day moving average with the color black:

```
> plot (PNG "aapl_20dayma.png") [Data2D [Title "AAPL - One Year, %
Change", Style Lines, Color Red] [] aapl252pc, Data2D [Title "AAPL 20-Day
MA", Style Lines, Color Black] [] aapl252ma20 ]
True
```

This gives the following chart:



Inspect the new image, `aapl_20dayma.png`. Note how the black line is now smooth compared to the original red line. The jaggedness is gone. The moving average line matches the original form of the share price, but without the noisiness to the data. The moving average function has been described as just that – a noise removal function of data. There are many algorithms in the field of pattern recognition that require the original dataset to have a noise removal function passed over the data as an initial step, and this is one such approach.

Does this chart help us to answer any questions? We can identify the brief moments when the stock is performing above its average stock value and when it isn't. Many investors will use two moving averages in their work (typically, a 50-day window and a 200-day window) and look for where the moving average lines cross. Investors will then pinpoint these crossings as an indicator of when they should buy or sell a stock.

This study of images and looking for patterns in the charts is a form of investing known as technical analysis. It is the opinion of this author that technical analysis is an attempt to predict the future of a stock price based on the history of that stock price when the reality is that they are mostly unrelated. Technical analysis should not be used as a substitute for traditional research.

Disclaimer: Your author doesn't own shares in any of the three companies mentioned in this chapter. Using the strategies demonstrated in this chapter, the reader can discover companies with a one-year growth percentage greater than 50 percent. While this practice of discovering high growth companies is exciting, remember to be skeptical of your findings. Don't fall into the trap of interpreting the past growth of a company as being indicative of its future growth. You can discover potential winners just as easily as losers with this method. Every company's fate hinges on tomorrow's good press or bad press. Predicting the past is easy. Predicting the future is not.

Plotting a scatterplot

We are going to close this chapter with a final visualization related to the earthquake dataset that we saw in *Chapter 2*, *Getting Our Feet Wet*. For this, I returned to the USGS page and downloaded their listing of earthquakes in the past month, totaling 7,818:

```
> convertCSVFileToSQL "all_month.csv" "earthquakes.sql" "oneMonth" ["time
TEXT", "latitude REAL", "longitude REAL", "depth REAL", "mag REAL",
"magType TEXT", "nst INTEGER", "gap REAL", "dmin REAL", "rms REAL", "net
REAL", "id TEXT", "updated TEXT", "place TEXT", "type TEXT"]
```

Using this dataset of earthquakes, we would like to plot the location of every earthquake in the world that happened in the path month. In this dataset, we have longitude coordinates (similar to an x-axis) and latitude coordinates (similar to a y-axis).

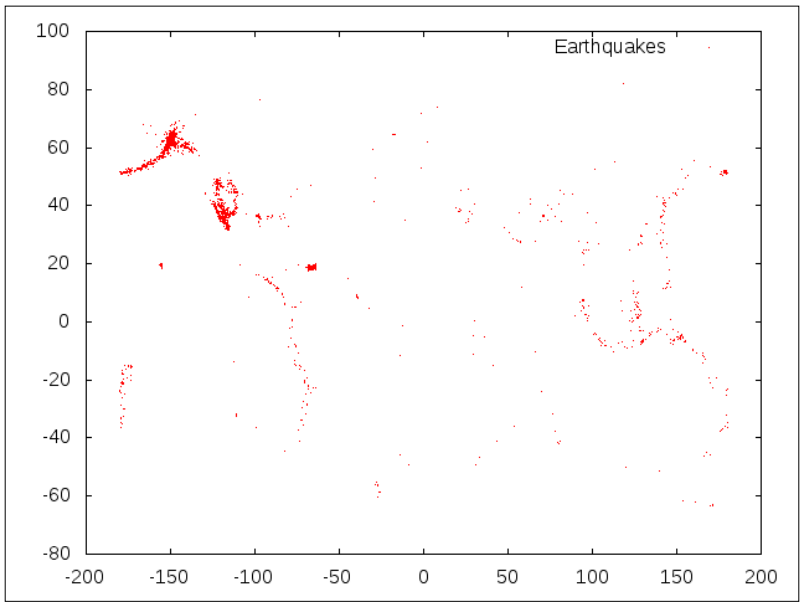
Let's craft a function to retrieve the latitude and longitude coordinates, similar to our `pullStockClosingPrices` function declared at the beginning of this chapter:

```
pullLatitudeLongitude :: String -> String -> IO [(Double, Double)]
pullLatitudeLongitude databaseFile database = do
  sqlResult <- queryDatabase
    databaseFile
    ("SELECT latitude, longitude FROM " ++ database)
    return $ zip (readDoubleColumn sqlResult 1)
                 (readDoubleColumn sqlResult 0)
```

Good. Now we should pull the coordinates of each earthquake. We will be plotting using the `Dots` style in EasyPlot:

```
> earthquakeCoordinates <- pullLatitudeLongitude "earthquakes.sql"
"oneMonth"
> coords <- pullLatitudeLongitude "earthquakes.sql" "oneMonth"
> plot (PNG "earthquakes.png") [Data2D [Title "Earthquakes", Color Red,
Style Dots] [] coords ]
```

This gives the following chart:



From this plot, we can make out the western coastline of North America and South America, as well as the Eastern coastline of Asia, Indonesia, the South Pacific Islands, and the Aleutian Islands. This is to be expected; these parts of the world get more earthquakes than other parts of the globe. We can also see giant areas of white; other parts of the globe don't see earthquakes that often.

Where are the strongest earthquakes? Better stated, how might we plot the earthquakes with a magnitude higher than 2 in the color blue and those earthquakes with a magnitude less than or equal to to 2 in red? That will be an exercise for the reader.

Summary

In this chapter, we've explored how to use the EasyPlot library, which allows us to interface the popular plotting utility gnuplot from within Haskell. The stock market is a wealth of information, and there is a long history of attempting to predict the stock market using plots of stock prices. Plots should help us to answer questions about data. Plots can also help us to explore data in the search of questions, if we are simply investigating data for the sake of it.

Our next chapter will help us to validate our assumptions about data through testing. We are going to be exploring data through statistics and Haskell.

5
Hypothesis Testing

Anyone can make claims and many actually do. The joy of data analysis lies partly in the ability to claim that a condition such as an event or a property has a measurable impact and defend that claim with a proper analysis. A claim that is testable is called a hypothesis. This chapter looks at the concept of hypothesis testing. In hypothesis testing, we examine a condition to determine whether it has a measurable impact on data or not. Since a hypothesis is only limited by one's imagination, many hypotheses will be determined as claims that do not have a measurable impact on the data. This doesn't mean that a claim isn't true. Sometimes, it just isn't measurable. For this reason, we stress that you should be skeptical about a hypothesis (which includes even your own hypothesis). Be prepared to have all of your ideas stated in a testable form and then perform the test.

Data in a coin

If you have a coin, you have money. If you flip that coin once and record the side that faces up, you have a result. If you flip that coin a second time and record the result, you have data.

Imagine that I have in my possession a fair coin. By the term *fair coin*, I mean that the coin has a side for heads, a side for tails, and (when flipped) the coin will land on the ground with a 50-50 chance of either heads or tails being the side that faces up (other possibilities, such as the coin landing on its edge, are not considered). You are allowed to inspect the coin and then hand it back to me. With the coin back in my possession, I cast a magic spell. I say a few magic words and tell you that the coin is now different. The coin is now in your hands and you are allowed to inspect it again. At a glance, the coin appears to be exactly the same as it was before I cast the spell. Did my spell have any magical effect on the coin? How will you be able to tell?

Hypothesis test

You will instantly realize that this is going to require a test. Our test is divided into two equal, yet opposite, outcomes:

- The magic spell did not have a measurable impact on the coin
- The magic spell did have a measurable impact on the coin

We will call the first outcome a null hypothesis, which is considered to be the default position. In plain English, a **null hypothesis** is the result that conveys that there was no change as a result of our action. I took the coin and said a few magic words, but whatever it is that I intended to do with this spell did not have a measurable impact on the coin. The second outcome is called an alternative hypothesis, which means that something did happen. My magic spell may have impacted upon the coin. There's also a possibility that I quickly shaved off a tiny edge of the coin when no one was looking to make the coin land on one side more often than the other. Our test will be able to indicate that a change was detected in the coin, but it will never be able to tell you the cause. Without performing any testing, our intuition should lean towards the idea that nothing happened and our skepticism should be rooted in the idea that something happened. Hence, the focus of our test will be the evaluation of the alternative hypothesis.

If we are not able to detect a change in our coin, we say that we *failed to reject the null hypothesis*. In other words, our test should attempt to reject the idea that no changes were made to the coin (otherwise, there will be no point in testing). If we are not able to detect a change, we *fail* in that rejection. There is an important distinction here that we need to make regarding the two cases where we may either *fail to reject* or *accept* the null hypothesis. When we are not able to detect a change that may have been made to the coin, it means that this particular test was unable to find a change that may have been made to the coin. There is a possibility that a future test can detect how the coin was changed. By stating that we fail to reject the null hypothesis, we leave open the possibility that a change happened, but we were not able to find it.

Establishing the magic coin test

"I know! I will flip the coin 1,000 times and count the number of times I see the coin land with heads face up."

"Good. What do you expect the number of heads to be?"

"Oh, I suppose it should be around 500."

This is the application of the exception formula. In the following formula, *X* is called a random variable. It is an event that will transpire with either a result of 1 (a coin lands on heads) or 0 (when a coin lands on tails or on the edge) based on the probability *p*. *N* represents the total number of event outcomes (which, in this case, is 2). The probability p_i represents an event's probability and X_i represents that event's value. Since there are two events, the first event has a probability of 0.5 and a value of 1, and the second event has a probability of 0.5 and a value of 0.

$$E[X] = \sum_{i=1}^{N} p_i \times X_i$$

We are going to treat heads in our example as an outcome that has a value of *1* and all other outcomes have a value of *0*. The probability of heads is *0.5* and tails is *(1 - 0.5)*, which is also *0.5*:

$$E[coinflip] = 0.5 \times 1 + (1 - 0.5) \times 0 = 0.5$$

Simply multiply *0.5* by 1,000 to get the expected number of 500 heads. However, you may realize that the true number might not be exactly 500 heads, but it should be close.

Understanding data variance

"We will set a window value. If the mean of the coin flips is within 500 plus or minus this window value, we will not be able to claim that the spell had no effect on the coin."

Sounds good. However, what should we set our window value to? We need to know how much our random variable spreads out over time. If we set the window value to 0, only an experiment with exactly 500 heads will pass. That might be too strict given that even a perfectly fair coin will not produce this value every time. If we set a window value of 50 (meaning plus or minus 50 heads), most people will agree that it is within the natural spread of a fair coin. If we set a window value of 100, we might still have a high number of people who believe that our coin was fair, but it should be fewer than when we set our window value to 50. If we set the window value to 300 (this means that we can have the total number of heads as low as 200 and as high as 800), it will be reasonable to conclude that this test is not useful to us. We need to understand how this variable spreads naturally. An understanding of this spread is called variance.

However, how do we evaluate this thought experiment without data? We need to study the problem itself. A Bernoulli trial is a single experiment (in this case, a coin flip) that results in a success (heads) or a failure (all other outcomes). When the trial is repeated, say, 5 times, we get sequences of heads and tails, such as HHTHT or THHHT. We can convert these sequences into values—11010 and 01110. Each sequence will have a computable total number of successes (in our small example, both trials resulted in a total of three successes) and these sets of experiments can be plotted into a histogram. This small example has 6 possible totals from 0 successes to 5 successes.

Probability mass function

We can illustrate this experiment by computing the average of all the possible sequences of 1,000 coin flips (of which there are 2^{1000} combinations (type `2 ^ 1000` into GHCi to see the size of this number). There is a better way. When many Bernoulli trials happen, the collection of these trials forms a binomial distribution. The following is a convenient formula to recreate the binomial distribution of data called the *probabilityMassFunction*. The plotting of the *probabilityMassFunction* will produce the same plot as a histogram of all the possible outcomes of coin flips. The formula for this can be denoted as:

$$probabilityMassFunction(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

This formula can be framed as a sentence in the following way; an event that succeeds with the probability *p* will succeed exactly *k* times out of *n* trials out of all possible outcomes with a probability determined by multiplying the probability of at least *k* desired successes by the probability of at least *(n-k)* desired failures by all of the possible arrangements that *k* number of successes can be arranged in *n* trials.

We can create a histogram of coin flips using Haskell. To make our lives easier, we are going to install the `Combinatorics` package. This packages contains the function that allows us to quickly perform the calculation required to determine the number of possible arrangements of *k* successes out of *n* trials:

```
$ cabal install exact-combinatorics
```

At the top of our `LearningDataAnalysis05.hs` file, make sure that our package is imported:

```
import Math.Combinatorics.Binomial
```

Next, let's craft the function:

```
probabilityMassFunction ::
    Integral a => a -> a -> Double -> Double
probabilityMassFunction k n p =
    (fromIntegral (n 'choose' k))
      * (p^k) * ((1-p)^(n-k))
```

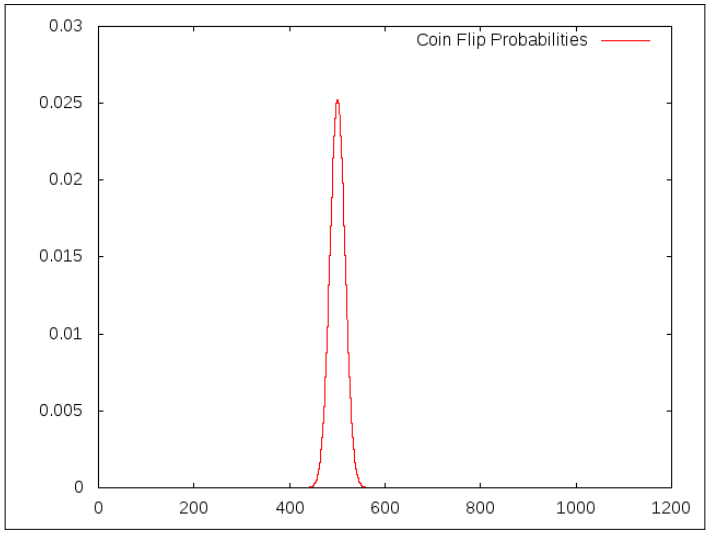
We will call the `plot` function (introduced in *Chapter 4*, *Plotting*) using the new `probabilityMassFunction` function to create a graph with a range of 0 to 1,000. To gain access to the `LearningDataAnalysis04` and `LearningDataAnalysis02` modules (used later in this chapter), use the following GHCi command:

```
> :l LearningDataAnalysis02 LearningDataAnalysis04 LearningDataAnalysis05


> :m LearningDataAnalysis02 LearningDataAnalysis04 LearningDataAnalysis05
```

Now, we will plot the function, as follows:

```
> import Graphics.EasyPlot> plot (PNG "coinflips.png") $
Function2D [Title "Coin Flip Probabilities"] [Range 0 1000] (\k ->
probabilityMassFunction (floor k) 1000 0.5)
```

The following screenshot shows the result of the preceding command:



Note that the data has a sharp peak at the 500 mark. It can be seen that the majority of the peak's width extends somewhere between 400 and 600. The portions of the graph that lie before and after the peak indicate that the probability of occurrence is almost 0. It is not impossible, but it is just highly unlikely. Let's find out the outcome of the most likely event at the peak of this plot:

```
> probabilityMassFunction 500 1000 0.5
2.52250181783608e-2
```

The mostly likely event from all the possible coin flips will happen 2.5 percent of the time. While this may seem small, it is still larger than the remaining 1,000 possible totals. While we are at it, take a note of the entire plot. The plot represents the probability of every individual possible outcome, no matter how insignificant. If we were to add up the probability of every possible outcome, the sum should be exactly 1. We will demonstrate this with Haskell:

```
> sum $ map (\k -> probabilityMassFunction k 1000 0.5) [0..1000]
1.0
```

The probability that any event will happen is equal to the sum of the probabilities of each event outcome, and this probability is always equal to 1.

Determining our test interval

We wish to create a test that will allow us to determine whether the results of the 1,000 enchanted coin flips fall within 99 percent of the possible fair coin outcomes. I picked 99 percent because the value needs to be convincingly high to show that the enchanted coin is different to a fair coin. This section of the process of testing is arbitrary. We need to know the range of events that result in the sum of all probabilities that equal 0.99 that reside at the center of the problem's mass function plot.

With a bit of trial and error, we will see that a window size of 40 hits our mark, which is close to the 99 percent threshold that we selected. I manually tested the ranges until I came across a range that is close to 0.99. It probably took about four tries to find a result that I liked:

```
> sum $ map (\k -> probabilityMassFunction k 1000 0.5) [(500-40)..
(500+40)]
0.9896118684338442
```

Establishing the parameters of the experiment

Finally, we can establish the parameters of our experiment. The null hypothesis is that the enchanted coin will flip heads 500 times (plus or minus 40). The alternative hypothesis is that the enchanted coin will flip heads more than 540 times or fewer than 460 times. Once we perform the experiment and gather the results, we will accept the alternative hypothesis if the result is greater than 540 or fewer than 460. Otherwise, we fail to reject the null hypothesis.

Introducing System.Random

We will now begin our actual experiment. To do this, we will not put any magical spells over real coins. We will use Haskell. Though Haskell is a purely functional programming language, it does have the ability to extend outside itself to look for resources, including the ones related to random number generation.

To use the `System.Random` module, you will first need to create a new random number generator, as follows:

```
> import System.Random
> g <- newStdGen
```

Using this new random number generator, we can utilize the `random` function to return pseudorandom values (and a new generator). The random number generation functions can output any numeric type we desire, so I have explicitly stated that I want a `Double` type. Note that if you make this call again a second time, you will get the same result thanks to Haskell's side effect-free nature. Here are the results that I get when I make a call to `random`:

```
> random g :: (Double, StdGen)
(0.8872828052781477,805351557 696985193)
```

We can use the `random` function to generate infinite random numbers in the range of 0 to 1. Since you probably do not need an infinite amount of random numbers, use the `take` function to limit this to your needs. Note that the first number in the list is identical to the previously generated random number. This is not a fluke. I didn't change my random number generation variable between these two calls. Here, we will generate three random values in the range of 0 to 1:

```
> take 3 $ randoms g :: [Double]
[0.8872828052781477,0.6612757244159314,0.7335027565852938]
```

We can also generate random values that are integers ranging from two specified values using the `randomRs` function. Again, you will need to use the `take` function to limit this to your needs. Here, we generate 5 random values in the range of 0 to 100:

```
> take 5 $ randomRs (0,100) g
[62,55,29,69,20]
```

Performing the experiment

Since our experiment calls for the flipping of a coin 1,000 times (where tails is equal to 0 and heads is equal to 1), we will use the `randomRs` function to generate 1,000 values on the integer range of 0 to 1 (the speaking of any magical words while issuing this line is optional):

```
> let coinflips = take 1000 $ randomRs (0, 1) g
```

What is the final result of this experiment? We will compute the sum of the `coinflips` value (your result may vary), as follows:

```
> sum coinflips
492
```

Since my result is within the range of 460 to 540, the experiment failed to reject the null hypothesis. There might have been some actual magic in my magical spell, but our experiment is not able to detect any change, and hence, the possibility is left open. If the experiment accepted the alternative hypothesis, it may mean that there is something unique to your system that justifies the result. It may also mean that you should re-run your experiment to be sure.

Does a home-field advantage really exist?

We perform tests on data for the following two reasons:

- We wish to evaluate a claim
- We have a limited amount of data

Both must be true in order to justify a test. If you have a complete picture of the data related to a claim, there will be no need to perform a test because a simple calculation will suffice. In the next example, we are going to have a look at an age-old claim in sports and use a limited amount of data to test the claim.

There is some important terminology that we will discuss here—population and sample. A complete picture of data is called a **population dataset**. Anything less than a complete picture is called a **sample dataset**. As discussed in *Chapter 1*, *Tools of the Trade*, sometimes the population dataset is so large that we are unable to work with it using the resources that are available on a single computer. A strategy that can be used to work with big data is to take a sample of the data that is small enough to be worked upon on a single computer and then test our claims using the limited dataset.

Baseball in America has a history dating back to the 19th century, and the data that we have on some teams is just as old. We shall continue our discussion of testing while using some historic professional baseball data. The Retrosheet website, `http://retrosheet.com/`, has many of these records available for viewing purposes in the CSV format. We will test the claim that a team scores more runs while playing at its home stadium as opposed to an away stadium. For this, we will need data (provided by Retrosheet), and we need to establish the null hypothesis and an alternative hypothesis using the following theory:

- The null hypothesis will be that there exists no difference in runs, or that a lesser number of runs were scored while playing at a home stadium as opposed to an away stadium
- The alternative hypothesis will be that more runs were scored while playing at a home stadium as opposed to an away stadium

I visited the Retrosheet website and found out that their page for game logs according to the year. Then, I downloaded the data corresponding to the 2014 dataset. At the time of writing this book, the page where you can download the game log data is `http://www.retrosheet.org/gamelogs/index.html`.

Converting the data to SQLite3

On downloading the file and unzipping its contents, you will quickly realize that there are many columns in the csv file with a huge amount of data. It will be fun to explore all of this data, but we are only interested in a few columns. To break this file into a few necessary columns, I used the `cut` gnuplot tool (with apologizes to my readers using Windows) to filter this data down to something far more manageable, as follows:

```
$ cut -d, -f 1,4,7,10,11 GL2014.TXT > winloss2014.csv
```

The `-d` flag tells `cut` that we are using a comma delimiter. The `-f 1,4,7,10,11` flag tells `cut` that we require the first (date), fourth (away team), seventh (home team), tenth (away runs), and eleventh (home runs) columns. The convenient aspect of baseball data is that any time a baseball game is described, the away team is always listed first, followed by the home team. If you don't have the `cut` tool, you can load the csv file into your favorite spreadsheet software, delete the unnecessary columns, and export the document back to the csv file type.

Once the dataset has been filtered down to just these five columns, we can import the data into a SQLite3 database using Haskell, as follows:

```
> convertCSVFileToSQL "winloss2014.csv" "winloss.sql" "winloss" ["date
TEXT", "awayteam TEXT", "hometeam TEXT", "awayscore INTEGER", "homescore
INTEGER"]
Successful
```

Exploring the data

We can quickly get an idea of the awayteam runs and hometeam runs using the `sum` function within the SQL query, as follows:

```
> queryDatabase "winloss.sql" "SELECT SUM(awayscore), sum(homescore) FROM
winloss"
[[SqlByteString "9791",SqlByteString "9966"]]
```

We can immediately see that there is a difference between the away team runs and the home team runs, and the home team runs scored more. Perhaps, there is some validity to the claim. First, we should get the performance of each team when they played at home. This is best done with SQL. With the following line, we will see the number of runs each team scored while playing at their own stadium. The `ORDER BY` clause will help us keep our teams in order, as follows:

```
> runsAtHome <- queryDatabase "winloss.sql" "SELECT hometeam,
SUM(homescore) FROM winloss GROUP BY hometeam ORDER BY hometeam"
> runsAtHome
[[SqlByteString "ANA",SqlByteString "362"],[SqlByteString
"ARI",SqlByteString "343"],[SqlByteString "ATL",SqlByteString "280"],...
remaining content clipped
```

This time, we will gather the performance of each team as they play stadiums away from home, as follows:

```
> runsAway <- queryDatabase "winloss.sql" "SELECT awayteam,
sum(awayscore) FROM winloss GROUP BY awayteam ORDER BY awayteam"

> runsAway

[[SqlByteString "ANA",SqlByteString "411"],[SqlByteString
"ARI",SqlByteString "271"],[SqlByteString "ATL",SqlByteString "293"],...
remaining content clipped
```

Use `zip` to get a pairwise comparison of each team. We will also use the `readDoubleColumn` function that was created in *Chapter 4, Plotting* as follows:

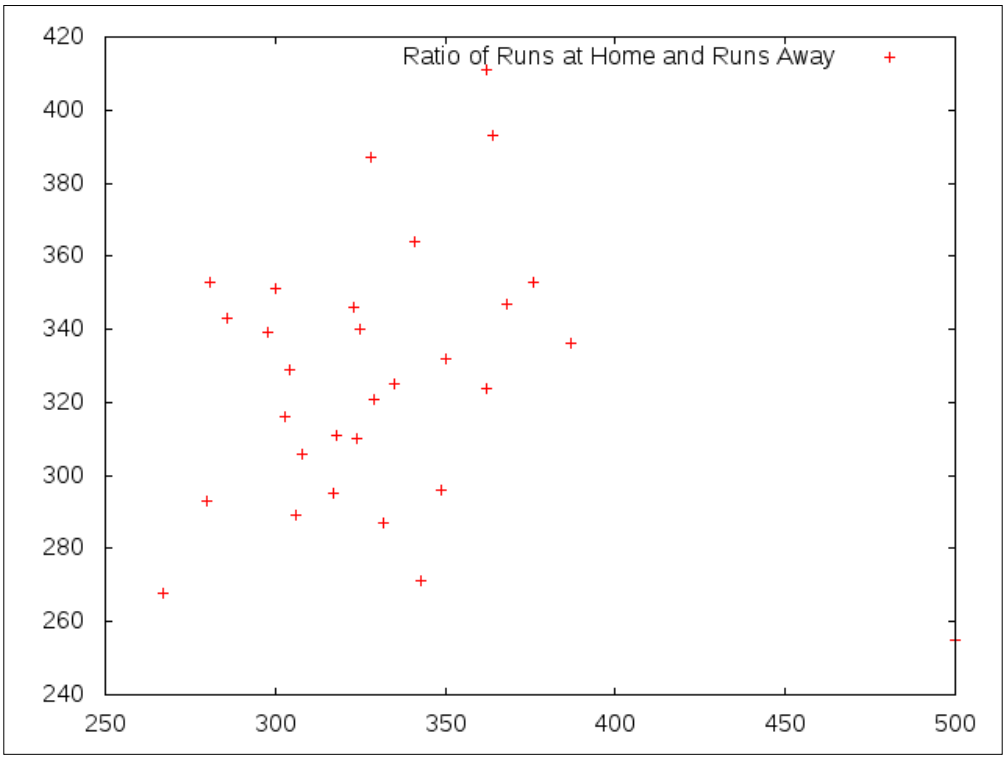
```
> let runsHomeAway = zip (readDoubleColumn runsAtHome 1)
(readDoubleColumn runsAway 1)

> runsHomeAway

[(362.0,411.0),(343.0,271.0),(280.0,293.0),(341.0,364.0),(324.0,310.0),(3
35.0,325.0),(308.0,306.0),(306.0,289.0),(323.0,346.0),(500.0,255.0),(364.
0,393.0),(318.0,311.0),(300.0,351.0),(328.0,387.0),(349.0,296.0),(329.0,3
21.0),(368.0,347.0),(304.0,329.0),(286.0,343.0),(376.0,353.0),(303.0,316.
0),(350.0,332.0),(267.0,268.0),(281.0,353.0),(325.0,340.0),(332.0,287.0),
(317.0,295.0),(298.0,339.0),(387.0,336.0),(362.0,324.0)]
```

Plotting what looks interesting

I bet that this is some interesting data. We will plot it, as follows:

```
> import Graphics.EasyPlot

> plot (PNG "HomeScoreAwayScore.png") $ Data2D [Title "Runs at Home (x
axis) and Runs Away (y axis)"] [] runsHomeAway

True
```

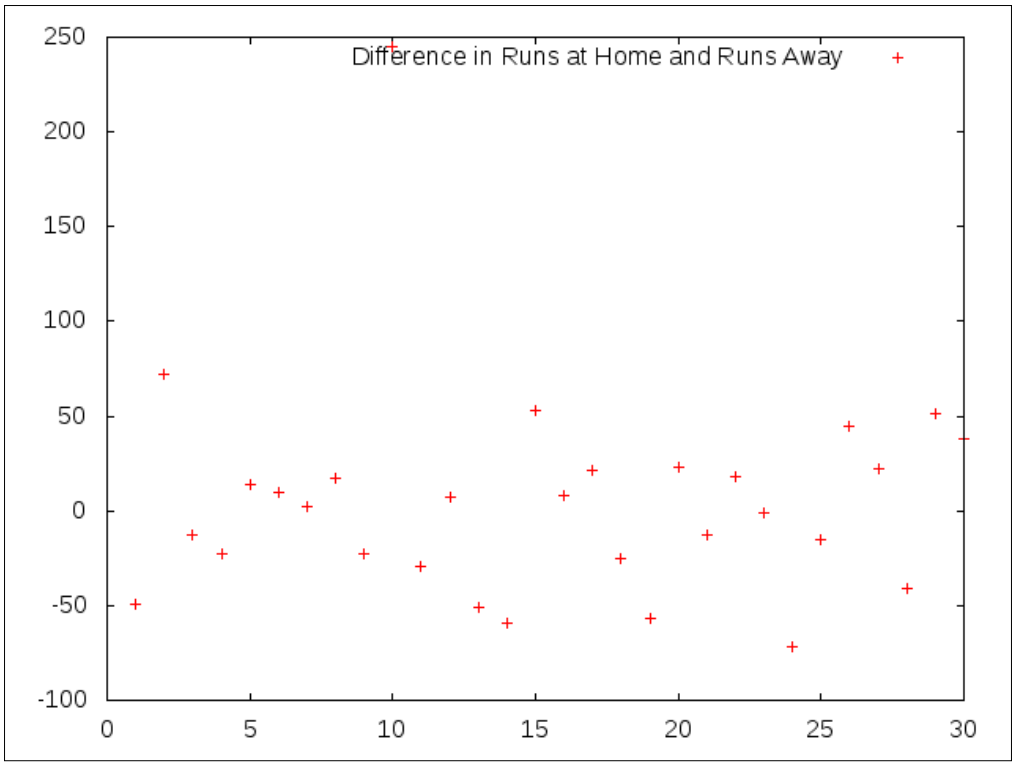
The following screenshot shows the result of the preceding command:



The preceding screenshot can be far more useful to us if we can identify the team that is associated with each data point (which is not something that the EasyPlot library currently allows). We will subtract the away scores from the home scores so that the data is focused on the scores that are around zero. Positive values represent more runs at home, and negative scores represent more away scores:

```
> let runsHomeAwayDiff = map (\(a,b) -> a-b) runsHomeAway
> plot (PNG "HomeScoreAwayScoreDiff.png") $ Data2D [Title "Difference in
Runs at Home and Runs Away"] [] $ zip [1..] runsHomeAwayDiff
True
```

The following screenshot shows the result of the preceding command:



You may note that there is a data point in this chart that represents almost 250 positive runs. This data point represents the Colorado Rockies team, which played at Mile High Stadium. This stadium is located a mile above sea level. Rockies aside, you will note that most of the data points range from -50 to 50 runs. Our goal is to figure out whether this data supports the claim that a team playing in its own stadium produces any scoring advantage. At a glance, it is still hard to tell from the picture.

Returning to our test

Now is a good time to restate our claim in mathematical terms, as follows:

- The null hypothesis is that the difference of runs scored at home and runs scored at the away games will be less than or equal to 0
- The alternative hypothesis is that the difference of runs scored at home and runs scored at the away games will be greater than 0

The next step is to compute the sample mean of this dataset. Since this is a sample dataset, we will call our mean a sample mean. Simply compute the average of each difference in runs for the listing of teams. We can easily compute the sample mean on a single machine. The population mean is a part of an unknown quantity that we will test. The following line requires you to keep the `LearningDataAnalysis02` module loaded:

```
> average runsHomeAwayDiff
5.833333333333333
```

Each team scored, on average, almost 6 more runs at home than away during the entire 2014 season. While this may seem like evidence to support our claim, we are not finished with the test.

The standard deviation

The **standard deviation** is a term that can be interpreted as the spread of the data. If the standard deviation is a small value, then the data is clustered around the mean. If the standard deviation is a large value, then data is scattered. We compute the standard deviation by finding the adjusted average squared distance from the average. We say that this is an adjusted average instead of just an average to take into account that this is a sample and not the entire population. To compute the adjusted average, we will divide the sum by the number of elements subtracted by 1. This *n-1* adjustment is known as *Bessel's correction*. The sample standard deviation is easily computed with the data sampled from the year 2014. The population sample deviation represents an unknown quantity.

We need to make sure the following `import` statements are found at the beginning of our `LearningDataAnalysis05.hs` file:

```
import Data.List
import Math.Combinatorics.Exact.Binomial
import LearningDataAnalysis02
```

```
standardDeviation :: [Double] -> Double
standardDeviation values =
    (sqrt . sum $ map (\x -> (x-mu)*(x-mu)) values) /sqrt_nm1
  where
    mu = average values
    sqrt_nm1 = sqrt $ (genericLength values - 1)
```

Good. We will now calculate the standard deviation:

```
> standardDeviation runsHomeAwayDiff
57.90365561286553
```

The sample mean is 5.83 runs and the sample deviation is 57.90 runs, which means that the average team scored 63.73 more runs at home or 52.07 more runs at away stadiums during the 2014 season.

The standard error

We will compute the standard error of the samples by dividing the standard deviation by the square root of the number of samples. The standard error is our way of expressing the mean to a precision:

```
> import Data.List
> standardDeviation runsHomeAwayDiff / (sqrt $ genericLength
runsHomeAwayDiff)
10.571712780392359
```

The standard error is 10.57 runs. We believe that the population mean of the runs' difference is 5.83 runs plus or minus 10.57 runs. Since 5.83 runs subtracted by 10.57 is -4.74 runs (and this is less than 0), we now have a small reason to doubt our original claim.

The confidence interval

So, is the population mean greater than 0 or not? We will attempt to answer this question with a confidence interval. A confidence interval is a way of expressing that a value is within two end points on a number line at a set confidence level. As with our last example, selecting the confidence level is arbitrary, but it needs to be a high value. In this example, we are going to use the 95 percent confidence level.

The example involving coin flips was contrived because it is possible to mathematically compute all the possible outcomes of 1,000 coin flips, and we showed how the outcomes followed a binomial distribution. In this example involving baseball data, we do not know the distribution of the data. In cases where the distribution of data is unknown, we have to assume a distribution. According to the central limit theorem, we can approximate a normal distribution with a defined mean and standard deviation, even when we aren't sure of the real distribution. The normal distribution has a curve that is similar to that of the binomial distribution. The normal distribution goes by another name that you are probably familiar with—the bell curve. We are going to assume that the difference in the runs at home to the runs at away stadiums follows a normal distribution.

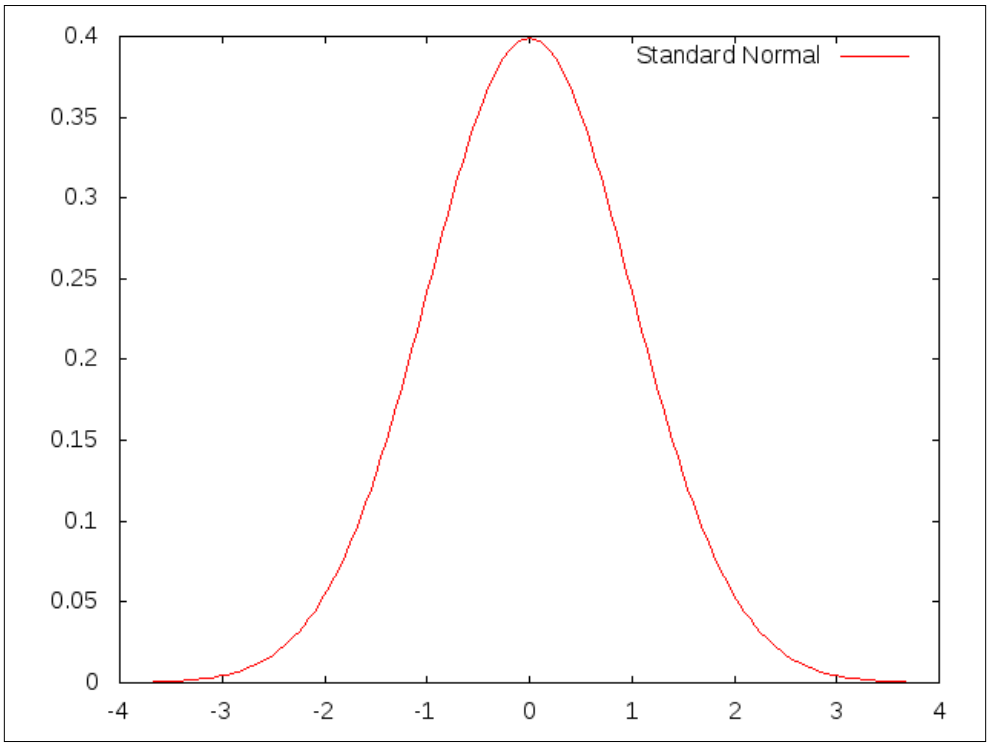
The standard normal distribution has a mean of 0 and a standard deviation of 1. This is the probability density function for the standard normal distribution. Similar to the binomial distribution, the integral of this function ranging from negative infinity to positive infinity is one. Its formula can be denoted as follows:

$$standardNormal(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$$

We will plot this formula on the range of -4 to 4. This function extends infinitely in both directions:

```
> plot (PNG "standardNormal.png") $ Function2D [Title "Standard Normal"]
[Range (-4) 4] (\x -> exp(-(x*x)/2)/sqrt(2*pi))
True
```

The following screenshot shows the result of the preceding command:



Based on our assumption that the difference in runs at home stadiums and away stadiums follows a normal distribution, we can establish a confidence interval that is centered around our sample mean. We must normalize this dataset so that we can utilize the standard normal curve. The normalizing of our dataset yields a distribution that is t-distributed. Just like what we did with the coin flip problem, we need to discover the interval that represents 95 percent of the area under the curve from the starting interval point to the ending interval point, and the center of this area is aligned to 0. The discovering of the area under the curve usually requires calculus, but there is a Haskell module that is designed just for this purpose, which makes the job a little easier.

We are going to work with the **Error Function** (**Erf**) module. You can install the Erf module using `cabal`, as follows:

```
$ cabal install erf
```

An introduction to the Erf module

Once this library is installed, you will have access to seven new functions that allow you to evaluate normal curves, including **normal cumulative density function** (**normcdf**) and **inverse normal cumulative density function** (**invnormcdf**). The `normcdf` function will take a parameter (*x*, which ranges from negative infinity to infinity) and returns the area under the curve of the normal standard starting from negative infinity to *x*. The `invnormcdf` function does the opposite of `normcdf`. Given a parameter (area, which ranges from 0 to 1), this function will return the value of *x* that produces a range from negative infinity to *x*, which has an area under the curve equal to the specified area.

For example, the mean of the standard normal curve is 0 (it is the center of the plot). The area under the curve to the left of 0 is 0.5 units. The area towards the right under the curve is also 0.5 units. Simply calling `normcdf 0` produces the area that is located towards the left of the plot, as follows:

```
> import Data.Number.Erf
> normcdf 0
0.5
```

We know that the total area under the curve is 1, which means that to produce the area to the right side of the plot of any value *x*, we must subtract `normcdf` *x* from `1`:

```
> 1 - (normcdf 0)
0.5
```

If we are interested in the value of *x* that produces half the area of the chart, we can use `invnormcdf` for this. We have already established that the value of *x* that designates exactly half the curve has to have the value of 0. So, it is no surprise that `invnormcdf 0.5` produces `0.0`:

```
> invnormcdf 0.5
0.0
```

Likewise, you can ask for the position of *x* that produces an area of 1, as follows:

```
> invnormcdf 1
Infinity
```

Haskell tells it like it is.

Using Erf to test the claim

We want to know the starting left position and the ending right position of a range that has an area of 0.95 (our confidence level), which is also centered at the middle of the standard normal distribution. This is going to be trickier than a simple `normcdf 0.95`. Because the range we desire will have 0.95 in the middle, we need to compute the area on both the sides of this range. Subtract `0.95` from `1` and divide the result by `2`:

```
> (1-0.95)/2
2.5000000000000022e-2
```

We must compute the position of *x* with an area of 0.025, as follows:

```
> invnormcdf 0.025
-1.9599639845400543
```

-1.96 will be the left side of our interval. Since we know that the area of the unused section towards the far right of the plot is also 0.025 units, the right side of the interval will have the value 1.96. The full interval at the 95 percent confidence level is -1.96 to 1.96. We can demonstrate this by calling `normcdf 1.96` and subtracting the area towards the far left tail of the curve from it:

```
> normcdf 1.96 - 0.025
0.9500021048517795
```

Next, we will multiply the standard error by each of the interval end points and add the mean of the sample data to it:

```
(5.83 + -1.96 * 10.57, 5.83 + 1.96 * 10.57) = (-14.89, 26.55)
```

A discussion of the test

The 95 percent confidence interval shows that, in the 2014 season, the teams score somewhere between -14.89 to 26.55 more runs at home stadiums than at away stadiums. As stated earlier, the null hypothesis is that the difference of runs scored at home and away games will be less than or equal to 0. Since our confidence interval clearly contains the 0 value in the range, we have failed to reject the null hypothesis. There might be a home-field advantage, but our sampling of the 2014 games failed to demonstrate this at the designated confidence level.

Summary

This chapter specifically looked at testing claims using the traditional statistical techniques and the Haskell `Erf` library. We defined the difference between the null hypothesis and the alternative hypothesis. We stressed that one should never accept the null hypothesis, but should always fail to reject it. We also emphasized a basic premise of skepticism—when faced with a claim that emphasizes how a particular event (magic spells, playing at home stadiums, new drugs, and so on) causes a change, your default position should be that the event doesn't change anything. You should also be willing to properly define a test that allows you to accept the alternative hypothesis and then perform the test.

Along the way, we explored binomial and normal distribution. Binomial distribution is used to compute the likelihood of discrete events at a known probability. The normal distribution is used as a reference to define confidence intervals. We also explored how Haskell handles pseudorandom number generation and used some generated numbers in a test.

The next chapter will cover regression analysis. Regression analysis serves two purposes. The first is to provide assistance in the discovery of correlations in data. The second is to determine data trends for the purpose of making predictions. We will explore both linear and nonlinear regressions on various datasets.

6
Correlation and Regression Analysis

In the previous chapter, we examined the classical procedure to test claims using the normal distribution curve. We also discussed the fundamental concept of variance and presented the function that computes the standard deviation of a dataset.

This chapter will examine the relationships between the input and output data. This is a truism to most sports fans—there exists a relationship between scoring and winning. It should seem obvious that sports teams that score higher points tend to win more games. As you might expect, the teams that don't often score high points tend to not win that often. The craft of measuring the relationship between the input (the number of points scored) and output (whether the team won or not) data is known as correlation analysis. Regression analysis allows us to estimate the result of an unknown output based on the input by creating an equation that minimizes errors between the independent and dependent variables that are believed to be linked. After creating a regression equation, we estimate the output for each of the known outputs based on the known inputs. This understanding is not without its drawbacks. A discussion of this approach should also come with an understanding of the potential errors that result from using it.

In this chapter, we will cover the following:

- The terminology of correlation and regression
- Study – is there a relationship between scoring and winning in baseball?
- Regression analysis
- The pitfalls of regression analysis

The terminology of correlation and regression

Before we explore any data, we will discuss some terminology. When using regression analysis, we need a set of input and output variables. Analysis where a single column of data is used as an input variable is known as **univariate analysis**. Analysis that takes multiple sets of input variables is known as **multivariate analysis**. Regression analysis allows us to estimate unknown values in a single column of output. The input data is known as an **independent variable**. There are no assumptions being made as to how independent variables behave. Unlike the input, the output is known as a **dependent variable**. The assumption in this case is that the input variables impact the output variable, for example, we will return to our assumption that scoring high points leads to a team winning more often than an average number of times. An average team in this context is a team that wins and loses an equal number of times. A team can work hard, practice, and exercise to their full potential in order to account for the offensive effort of the team, which will account for the team scoring more points than the average number of points. We generally believe that these actions translate to a team that wins more often than an average team (while we ignore some equally important factors like the defensive abilities of an opposing team). Likewise, a team puts forth a minimal amount of effort into practice in order to score fewer points. We also believe that these actions translate to a team that wins less often than an average number of times. In both cases, the team represents an independent variable. The amount of hard work put towards an offensive effort is purely under their control. Unlike offensive effort, winning is not completely under their control. An opposing team might have a better offensive or defensive plan, or more capable players on their side. These qualities represent an unknown variable when performing regression. Finally, there is luck. Ignoring luck, our assumption is that winning is dependent on scoring, and scoring is dependent on offensive effort.

The expectation of a variable

An independent variable is denoted as X. If we have multiple independent variables, then you will see each independent variable denoted as X_1, X_2, X_3, …, X_m. A dependent variable is denoted as Y. The X and Y variables represent a dataset of n values (or observations). The mean of X is known as X-bar. If we wish to take every value of X and subtract X-bar from it, we will write it in the following way:

$$X - \bar{X}$$

The term *average* (which is the result of a sum of a listing of values divided by the number of values) also goes by the term *mean* as well as *expectation*. Sometimes, we will see the preceding formula written as follows:

$$X - E[X]$$

You can think of *E* as a function that computes the average of a list of values.

The result of this operation will be a new dataset with the average being `0`. If a value in this dataset is positive, we know that the result is above average. Likewise, negative values will give the result as below average.

The variance of a variable

We would like to know the spread of this variable. To do this, we will take each value in the dataset, subtract the mean, and then square the result. This produces a dataset that consists of positive values. After this step, add up all of the values and divide the sum by the number of observations to get the average squared distance from the average. This, as we discussed in *Chapter 5*, *Hypothesis Testing*, is the population variance. To find out the sample variance, rather than dividing by *n*, we divide it by *(n-1)*:

$$Var[X] = \frac{1}{n}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2$$

We can write the same formula with the help of a cleaner notation by using the *E* function:

$$Var[X] = E\left[\left(X - E[X]\right)^2\right]$$

To find out the population standard deviation, we will take the square root of the population variance. The standard deviation is signified by the Greek letter, sigma (denoted as σ):

$$\sigma = \sqrt{E\left[\left(X - E[X]\right)^2\right]}$$

Normalizing a variable

Now that we know the average distance from the average of the values in the dataset, we can use this to normalize the *X* variable. To do this, we will divide each value in *X*- *X*-bar by the standard deviation:

$$\frac{X - \bar{X}}{\sigma}$$

This normalized version of our dataset still contains positive and negative values, but it is also a measure of how extreme in distance the normalized variable is from the mean. A score between -1 and 1 means that a value is closer to the mean than the typical data value. Scores ranging from -2 to -1 and from 1 to 2 mean that the value is between one and two times distant from the mean than the typical value. Most of your data should fall between a score of -2 and 2. Scores from -3 to -2 and from 2 to 3 indicate that the value is a little more distant. A value greater than 3 or less than -3 means that this value is more than 3 times the distance from the average of the values in the dataset than the typical value. Values with a score in this range are considered rare and indicate special circumstances that merit investigation. Values that deviate significantly from the majority of a dataset are called **outliers**. When an outlier value has been identified, it possibly represents special circumstances and should be investigated for unique qualities. It can also mean something less special—a noisy data point was not properly identified in the cleaning phase of data analysis.

The covariance of two variables

When working with two variables (in our case, an input and an output variable), we may want to study how the variables move in conjunction with each other. Like variance, the tool known as covariance helps us to measure how variables relate to each other. Instead of one *X* variable, we now have two—*X* and *Y*.

We will begin by subtracting the mean of *X* from each value of *X*, whose answer is then multiplied by the result of the mean of *Y* subtracted from each value of *Y*:

$$\left(X - \bar{X}\right)\left(Y - \bar{Y}\right)$$

Again, we add each of these distance measures and divide the sum by the number of observations to find out the population covariance coefficient. To find the sample covariance coefficient, we change *n* to *n-1*:

$$Cov[X,Y] = \frac{1}{n}\sum(X-\bar{X})(Y-\bar{Y})$$

Again, we can write the same formula by using the *E* notation:

$$Cov[X,Y] = E\left[(X-E[X])(Y-E[Y])\right]$$

The covariance coefficient is a measurement of how the variables relate to each other. If the *X* variable increases as the *Y* variable increases, then this coefficient will have a positive value. Likewise, if *X* or *Y* increases and the other variable decreases, then this coefficient will have a negative value.

Finding the Pearson r correlation coefficient

We can normalize this value as we did before. This normalized value will always have a value from -1 to 1. We need the standard deviations of *X* and *Y* (σx and σy). This normalized version of the covariance value is known as the **Pearson r correlation coefficient**. The formula for this can be denoted as follows:

$$r = \frac{Cov[X,Y]}{\sigma_X \sigma_Y}$$

As with the correlation coefficient, a positive *r* value informs us that the variables are linearly correlated (as the value of one variable increases, the value of the other increases) and a negative *r* value informs us that the variables are inversely correlated (as the value of a variable increases, the other decreases). The closer an *r* value is to the extremes of -1 or 1, the more the strength of the correlation it indicates. An *r* value that is close to 0 tells us that the connection between the two variables is weak.

Finding the Pearson r^2 correlation coefficient

The *r* value is tweaked further to r^2 (we simply multiply r by itself). Because r ranges from -1 to 1, the value of r^2 will always be a value from 0 to 1. A higher r^2 implies that there is stronger evidence of a correlation. A lower r^2 implies little or no correlation. An r^2 that is greater than 0.9 is considered to be an excellent correlation, while an r^2 that is less than 0.5 is considered to be weak. Interpreting an r^2 is a form of art—it is what you make of it. The r^2 values go by the name, coefficient of determination.

We will not continue this discussion without talking about a key pitfall of r^2—the discovery of an input and output variable that produces a high r^2 value does not automatically imply that the input has an impact on the output. This is known by the saying, *correlation does not imply causation*. When data mining for correlations, you are bound to find pairs of variables that produce high r^2 values. Some of these input variables might have a causal effect on the output variable, and at times, you may find a similar pattern between two unrelated variables. This frequently happens when two output variables are compared. *This measurement only tells you what correlates, not whether the discovered correlations are sensible*. You might discover a link between the sale of food and the sale of beverages at a restaurant, but there is nothing interesting about this correlation because these variables are dependent on the number of customers who frequent the restaurant.

Translating what we've learned to Haskell

We can express these formulas to Haskell in the following way:

```
module LearningDataAnalysis06 where

  import Data.List
  import Graphics.EasyPlot
  import LearningDataAnalysis02
  import LearningDataAnalysis04
  import LearningDataAnalysis05

  {- Covariance -}
  covariance :: [Double] -> [Double] -> Double
  covariance x y = average $ zipWith (\xi yi -> (xi-xavg) * (yi-
  yavg)) x y
    where
      xavg = average x
      yavg = average y
```

```
{- Pearson r Correlation Coefficient -}
pearsonR :: [Double] -> [Double] -> Double
pearsonR x y = r
  where
    xstdev = standardDeviation x
    ystdev = standardDeviation y
    r = covariance x y / (xstdev * ystdev)

{- Pearson r-squared -}
pearsonRsqrd :: [Double] -> [Double] -> Double
pearsonRsqrd x y = pearsonR x y ^ 2
```

Study – is there a connection between scoring and winning?

We will continue using the baseball dataset that we introduced in the last chapter. We can check whether there is a correlation between scoring and winning. To do this, we will compute the runs scored per game for each team as well as the win percentage for each team. We must compute the total number of runs scored, the total number of wins, and the total number of games played by each team.

A consideration before we dive in – do any games end in a tie?

Before we can proceed with our analysis, we need to make sure that we handle data that does not quite fit into our model. In this case, we are talking about tie games. There are many games played each year and, often, games have an extra inning. However, does any game ever end up as a tie? A few select queries will let us know whether the home score equals the away score. The first query will tell us the number of games in our database and the second will report the number of games that ended in a tie:

```
> queryDatabase "winloss.sql" "SELECT COUNT(*) FROM winloss"
[[SqlByteString "2429"]]
> queryDatabase "winloss.sql" "SELECT COUNT(*) FROM winloss WHERE
awayscore==homescore;"
[[SqlByteString "0"]]
```

Games that ended in a tie are not a part of this dataset. All the 2,429 games of the 2014 season had clear winners.

Compiling the essential data

You may recall that the data is organized according to the individual games. For each record in the database, the away team is listed before the home team. We are going to split the information gathering phase into two steps— a step to gather the home team stats and a step for the away team stats. The information that we are going to gather is the 3-letter code for the team name, the number of home wins, the total number of runs scored at home games, and the number of home games. First, we will gather the stats related to the home team. The `GROUP BY` statement will make sure that all the teams stay in the alphabetical order, according to the team code:

```
> homeRecord <- queryDatabase "winloss.sql" "SELECT homeTeam,
SUM(homescore > awayscore), SUM(homescore), COUNT(*) FROM winloss GROUP
BY homeTeam;"
```

Second, we will collect the stats pertaining to the away team. With the following query, we will switch from looking at the information related to the home field to the data related to the away field:

```
> awayRecord <- queryDatabase "winloss.sql" "SELECT awayTeam,
SUM(awayscore > homescore), SUM(awayscore), COUNT(*) FROM winloss GROUP
BY awayTeam;"
```

Now, we must combine these datasets. First, we will gather the total number of wins. In the two matrices, this information appears in column index `1`:

```
> let totalWins = zipWith (+) (readDoubleColumn homeRecord 1)
(readDoubleColumn awayRecord 1)

> totalWins

[98.0,64.0,79.0,96.0,71.0,73.0,73.0,76.0,85.0,66.0,90.0,70.0,89.0,93.0,77
.0,82.0,70.0,84.0,79.0,88.0,73.0,88.0,77.0,87.0,88.0,90.0,77.0,67.0,83.0,
96.0]
```

Next, we will gather the total number of runs scored. This information appears in column index `2`:

```
> let totalRuns = zipWith (+)  (readDoubleColumn homeRecord 2)
(readDoubleColumn awayRecord 2)

> totalRuns

[773.0,614.0,573.0,705.0,634.0,660.0,614.0,595.0,669.0,755.0,757.0,629.0,
651.0,715.0,645.0,650.0,715.0,633.0,629.0,729.0,619.0,682.0,535.0,634.0,6
65.0,619.0,612.0,637.0,723.0,686.0]
```

Finally, we will gather the total number of games played by each team. There are 162 games in a typical baseball season, but we should not depend on this tidbit of knowledge and should instead calculate it ourselves. This information is in column index `3`:

```
> let totalGames = zipWith (+) (readDoubleColumn homeRecord 3)
(readDoubleColumn awayRecord 3)

> totalGames

[162.0,161.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,
162.0,161.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,162.0,1
62.0,162.0,162.0,162.0,162.0,162.0]
```

As you can see, almost every team played 162 games (save two teams that played 161).

We are still not done with the compiling of information. We now need the win percentage and the number of runs per game, both of which can be obtained by dividing `totalWins` and `totalRuns` by `totalGames`, as follows:

```
> let winPercentage = zipWith (/) totalWins totalGames

> let runsPerGame = zipWith (/) totalRuns totalGames
```

Searching for outliers

We mentioned previously that a simple definition of an outlier is a data point that is more than 3 standard deviation units from the mean of the dataset. Let's explore these two datasets (`runsPerGame` and `winPercentage`) for outliers. First, we will look at the `runsPerGame` function. Here, we will take the absolute value of the normalized dataset and check to see whether any of these values exceed `3` units:

```
> any (\xi -> abs( (xi - average runsPerGame) / standardDeviation
runsPerGame) > 3) runsPerGame

False
```

None of the values in the `runsPerGame` function exceed `3` units. Let's do this again for the `winPercentage` dataset:

```
> any (\xi -> abs((xi - average winPercentage) / standardDeviation
winPercentage) > 3) winPercentage

False
```

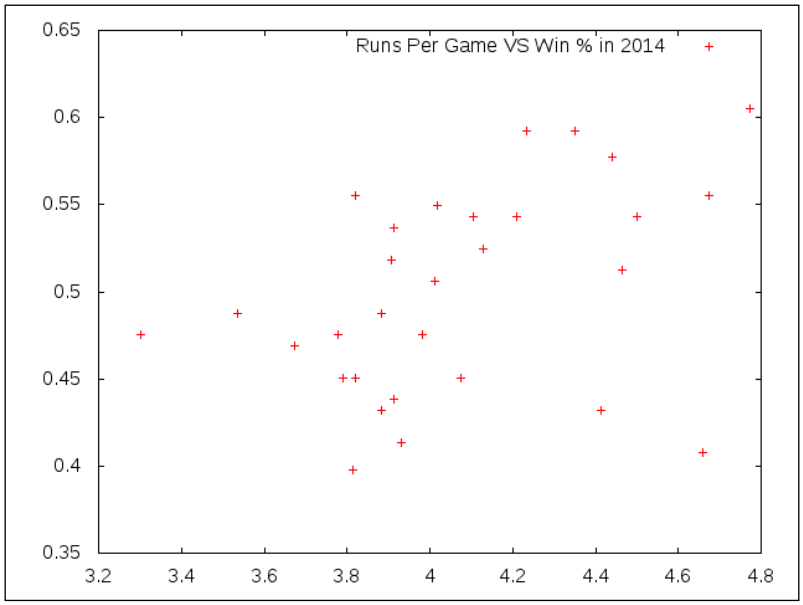
Again, none of the values in either the `runsPerGame` dataset or the `winPercentage` dataset seem to indicate that they contain an outlier. If we were to encounter an outlier, it might indicate that either something is wrong with the dataset, or there is something unique about that particular data point. Either way, outliers will require further investigation on the part of the data analyst. If you believe that an outlier in your data exists based on certain unique circumstances, you will be forgiven if you exclude this observation from your overall analysis and make a note of the exclusion.

Plot – runs per game versus the win percentage of each team

Let's plot this information. Plotting will prove useful during the interpretation of the results. The x-axis and y-axis of this chart will represent the number of runs per game and the win percentage of each team respectively:

```
> import Graphics.EasyPlot
> plot (PNG "runs_and_wins.png") $ Data2D [Title "Runs Per Game VS Win % in 2014"] [] $ zip runsPerGame winPercentage
True
```

The preceding statement would give the following chart as a result:



What immediately pops out when we look at this image is how cloudy the data looks. We can see that the data does trend upward, and this is evidence of a positive correlation. Yet, it is hard to make out a line in the data. You might notice that at the bottom right corner of the image is a data point representing the team with the third highest number of runs per game in 2014, and yet they have a dismal win percentage of less than 0.45. This (again) is the Colorado Rockies, who tend to have high scoring games at their one-mile-above-sea-level stadium, but this does not translate into a majority of wins for the team. At the opposite corner of this graph, we can see that there are three teams that scored less than 4 runs per game in more than half of their games in 2014. These are the teams with winning seasons in spite of having a less-than-stellar offense. This contradicts the idea that higher scores lead to the winning of more games.

Performing correlation analysis

Let's measure the correlation between these two variables:

```
> pearsonR runsPerGame winPercentage
0.40792278544575666
> pearsonRsqrd runsPerGame winPercentage
0.16640099888582482
```

A Pearson r score of 0.41 is positive, which indicates that there is a positive correlation between the runs per game and the win percentage. When you square this value to compute the r^2 score, you will see that it is 0.17. As we mentioned earlier, an r^2 score of less than 0.5 is considered a weak correlation of the two variables. We see in this analysis that scoring is a part of winning, but the relationship between these two variables should be considered weak. In other words, the idea that a team will improve their win percentage only by improving the runs scored per game is considered to be weak.

Regression analysis

Should we tell our coaches that scoring is not important? Of course not. A team needs to score at least one run to have a chance of winning a game. We should communicate to our coaches the importance of scoring more runs per game, even when we know that there is a weak correlation between scoring and winning. We communicate this importance by using regression analysis. With regression analysis, we create an equation that will allow us to estimate the win percentage of a team based on their runs per game value.

The approach that we will follow is known as **simple linear regression**. Linear regression is the simplest type of regression. The working assumption is that our data forms a straight line. While we admit that it is difficult to make out a line in our data, we shall make the assumption that a line exists. This line indicates the increase in the win percentage of a team as the team scores more runs per game. When one factor goes up, the other goes up linearly.

The regression equation line

A linear equation is as follows:

$$y = A + Bx$$

In this equation, x represents the number of runs per game, B represents the slope (or the gradient) of the line this equation gives, A represents the y-intercept, and y represents the estimated winning percentage of a team that is able to score x runs per game. The equation line will represent a best fit that will closely follow down the middle of the data, minimizing the difference in the distance between the points above and below the line.

Estimating the regression equation

The regression equation is a best fit line. The goal in crafting this equation is to produce the smallest overall error between the real data and what the equation will estimate the data to be. We can minimize the error term by computing the covariance of the X and Y variables and dividing it by the variance of X:

$$B = \frac{Cov[X,Y]}{\sigma_x^2} = \frac{Cov[X,Y]}{Var[X]}$$

We can compute the value of A (our y-intercept) by computing the average of X and Y and substituting these values into our linear equation:

$$A = \overline{Y} - \overline{X}B$$

Translate the formulas to Haskell

In Haskell, the preceding formula will look like the following code:

```
{- Perform simple linear regression to find a best-fit line.
  Returns a tuple of (gradient, intercept) -}
linearRegression :: [Double] -> [Double] -> (Double, Double)
linearRegression x y = (gradient, intercept)
  where
    xavg = average x
    yavg = average y
    xstdev = standardDeviation x
    gradient = covariance x y / (xstdev * xstdev)
    intercept = yavg - gradient * xavg
```

Here, I have renamed *B* to gradient and *A* to intercept.

Returning to the baseball analysis

When we execute the preceding code with our baseball data, we get the following output:

```
> let (gradient, intercept) = linearRegression runsPerGame winPercentage
> gradient
6.702070689192714e-2
> intercept
0.22742671114723823
```

The value of the slope is 0.07 and the value of the intercept is 0.23. A slope of value 0.07 indicates that if a team increases their runs per game by 1 run per game, it should increase their seasonal win percentage by an estimated seven percent. This percentage grows with a linear progression, so an increase of 2 runs per game will increase their win percentage by an estimated 14 percent (and so forth). Here, we will estimate the win percentage for a fictional team that was able to score `3`, `4`, and `5` runs per game for an entire season:

```
> 3*gradient+intercept
0.42848883182301967
> 4*gradient+intercept
0.4955095387149468
> 5*gradient+intercept
0.5625302456068739
```

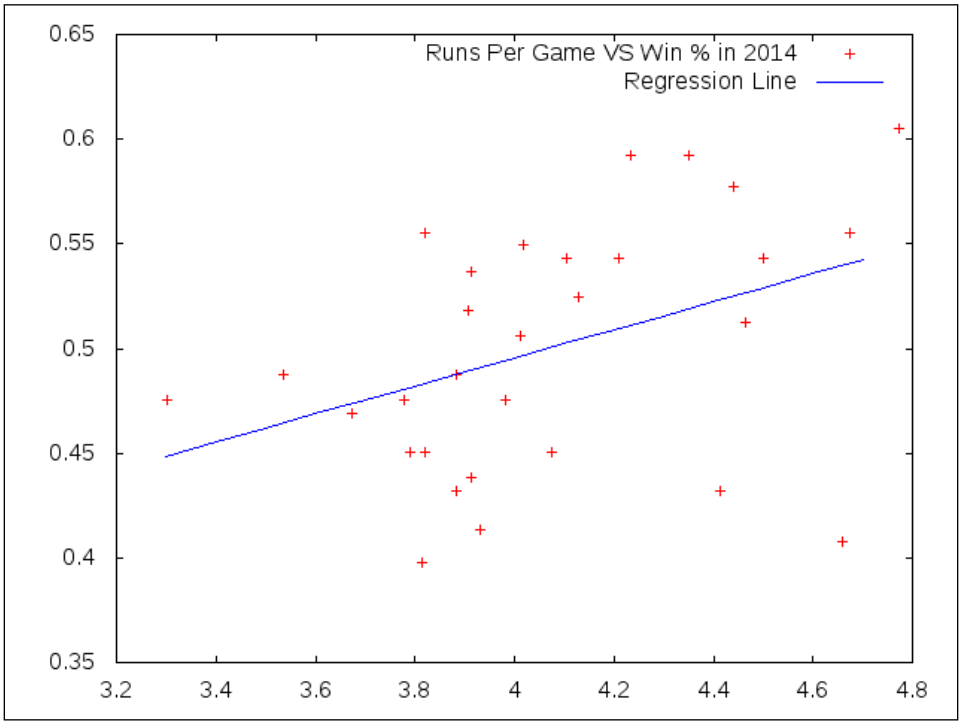
A team that scores 3 runs per game should win about 43 percent of their games. At 4 runs per game, the estimated win percentage is 50 percent. At 5 runs per game, this increases to 56 percent. In our dataset, the team with the highest win percentage won over 60 percent of their games while not quite hitting 5 runs per game.

Plotting the baseball analysis with the regression line

In the previous chart that displays the runs per game and the win percentage for each team, the chart ranges between 3.2 and 4.8 runs per game. I created a new dataset of line estimates based on the values of runs per year that range from 3.3 to 4.7. This way, we have a line that can fit nicely within the existing chart:

```
> let winEstimate = map (\x -> x*gradient + intercept) [3.3, 3.4 .. 4.7]
> let regressionLine = zip [3.3, 3.4 .. 4.7] winEstimate
> plot (PNG "runs_and_wins_with_regression.png") [Data2D [Title "Runs Per Game VS Win % in 2014"] [] (zip runsPerGame winPercentage), Data2D [Title "Regression Line", Style Lines, Color Blue] [] regressionLine]
```

The preceding statements would give the following chart as a result:



What makes the regression line so helpful in our analysis is that it allows us to quickly identify the teams that are above average and below average. Above the line exists the teams that did better than our estimate suggests. Below the line exists the teams that did worse than our estimate.

The pitfalls of regression analysis

There are several pitfalls of regression analysis. We will go over some of the limitations of this analysis:

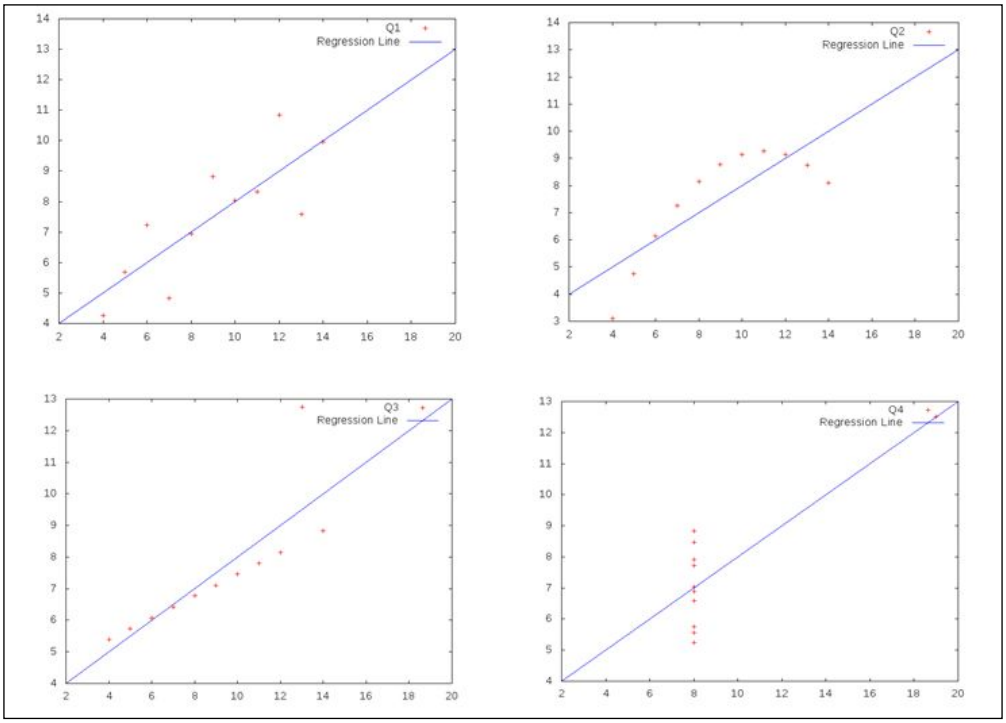
- **Keep your estimates close to the original input variable range**: The regression equation is in the form of a straight line. A line in a two-dimensional plane has a slope and a y-intercept and extends infinitely in two directions. Because of this, we are capable of estimating values beyond the range of our input variables, for example, the y-intercept of our equation is 0.22. This means that a baseball team that scores an average of 0 runs per game should win an estimated 22% of their games (which is laughable). Just because we can estimate values outside of the range of our input, it doesn't mean that we should.
- **There is more to regression than simple linear regression**: This chapter only looks at simple linear regression, but there are several other types of regression, including **log regression** (where we first compute the natural log of the output variable) and **log-log regression** (where we first compute the natural log of both the input and the output variables). The fundamental approach is the same, but having an understanding of the curvature of the data (and not automatically assuming that the data forms a line) sometimes yields better results.
- **Any analysis that uses the mean of a dataset is easily skewed**: Since the regression line is so dependent on the average of both the input and output variables, using simple linear regression can create a line that is distorted based on a single heavily skewed data point.

  The next example illustrates this clearly. The following table represents a contrived dataset known as an **Anscombe's quartet**. This dataset was manually developed so that each dataset has a nearly identical mean of the `x` column, mean of the `y` column, correlation coefficient, and linear regression line. It demonstrates the problem of simple linear regression. The procedure is not robust with respect to outlier data values or data that isn't in a linear order. In each of these four datasets, a simple linear regression analysis reports that all four datasets follow the same linear path when it is clear that three of them do not.

The data for the following table was taken from the Wikipedia entry for an Anscombe's quartet:

I		**II**		**III**		**IV**	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Plotting the data of this table would give us the following charts:



These graphs were created using the Haskell functions, the EasyPlot library that was defined in this chapter, and the same procedure that was used to analyze the baseball data (except that all the outlier observations were retained for the purpose of this demonstration).

Summary

In this chapter, we took a crash course in univariate analysis. We took a closer look at the ways used to compute covariance, the Pearson r score, and the Pearson r^2 score, as well as the methods deployed to interpret this data. We then took a look at the baseball dataset again and explored the question, *Is there a relationship between scoring and winning?* It may be a surprise to the reader, but we found out that the relationship between the two variables is weak. We also looked at regression analysis, which allows us to estimate unknown output variables based on the covariance of the existing data. We also spent time knowing about the pitfalls of blindly using simple linear regression.

The next chapter looks more at prediction, but this time from the perspective of Bayesian analysis. Bayesian probability is a form of conditional probability where we estimate the likelihood of events happening based on the evidence of past probabilities.

7
Naive Bayes Classification of Twitter Data

In the last chapter, we looked at regression analysis and created a simple linear regression based on baseball data in order to determine if a team was playing above expectations or below expectations. In some sense, we created a classifier. We took raw data and partitioned the data space into two classes, those performing above and those performing below expectation based on a linear interpretation of the data.

Up to this point in the book, all of the data that we have investigated has been numerical data. Looking at numerical data is great for an introduction to data analysis because so much of an analyst's job is applying simple formulas (such as average, normalization, or regression) to data and interpreting the results. Numerical data is only half the picture. Raw, unstructured data is an equally important that we haven't touched upon. We will look at unstructured data in this chapter.

In this chapter, we cover the following:

- An introduction to Naive Bayes classification
- Downloading tweets via the Twitter API
- Creating a database to collect tweets
- Analyzing text data based on word frequency
- Cleaning our tweets
- Creating a Naive Bayes classifier to detect the language of tweets
- Testing our classifier

In this chapter, we look at tweets, which are the short messages posted to the enormously popular social networking website, Twitter. Tweets only have one limitation; they must fit into 140 characters. Because of this limitation, tweets are short bursts of ideas or quick messages to other members. The collection of tweets by a member is called the *timeline*. The social network on Twitter follows a directed graph. Members are encouraged to *follow* other members and members who gain a follower have the option to follow back. The default behavior of each Twitter account is that no special permissions are required for a member to be followed, to view a member's timeline, or to send public messages that are delivered to another member. This openness allows members to have conversations with others who may join and leave at their pleasure. Active conversations can be discovered through the use of hashtags, which is a way of tagging a tweet with a subject, thus encouraging discussion on a topic by allowing others to find tweets in an easier manner. Hashtags begin with a # and are followed by a subject term. Searching hashtags is a way for people to quickly find like-minded members and build communities around an interest. Twitter is a social network with broad international appeal and supports a wide variety of languages. We wish to use Twitter in this chapter to study language specifically. Language is a tool that humans use to convey ideas to other humans. As the case may be, not everyone communicates in the same language. Look at the following two sentences:

- My house is your house
- Mi casa su casa

Both of these sentences convey the same idea, just in a different natural language (the first, obviously, in English, and the second in Spanish). Even if you don't know Spanish (I must confess that I don't), you might be able to guess that the second sentence represents a sentence in the Spanish language based on the individual words. What we would like to accomplish in this chapter is the creation of a classifier that will take a sentence and produce what it thinks is the best guess of the language used. This problem builds upon data analysis by utilizing that analysis to make an informed decision. We will attempt to gently step into the field of machine learning, where we attempt to write software that makes decisions for us. In this case, we will be creating a language detector.

An introduction to Naive Bayes classification

The Bayes theorem is a simple yet efficient method of classifying data. In the context of our example, tweets will be analyzed based on their individual words. There are three factors that go into a Naive Bayes classifier: prior knowledge, likelihood, and evidence. Together, they attempt to create a proportional measurement of an unknown quality of an event based on something knowable.

Prior knowledge

Prior knowledge allows us to contemplate our problem of discovering the language represented by a sentence without thinking about the features of the sentence. Think about answering the question blindly; that is, a sentence is spoken and you aren't allowed to see or hear it. What language was used? Of all of the tens of thousands of languages used across time, how could you ever guess this one? You are forced to play the odds. The top five most widely spoken languages are Mandarin, Spanish, English, Hindi, and Arabic. By selecting one of these languages, you have improved your odds (albeit still in the realm of speculation). The same basic strategy is taken by the Naive Bayes classifier; when stumped between multiple classes and the data presented is equally split, it leans towards the more popular category.

Prior knowledge is denoted in the following way:

$$P(A)$$

In other words, without any information to go on, the probability that a language will be selected will be based on the popularity of that language. This quantity is sometimes called the *prior belief*, since it can be difficult to measure. Care should be taken while estimating the prior belief since our results will be sensitive to this quantity.

Likelihood

Likelihood asks for the probability of our known features, given that the class of the features is already known. In essence, we ask how likely a tweet is represented by a particular language based on a single word in which we already know how likely we are to see that word in phrases representing this language. A word is a feature of a language. Some words are used often, such as the word *the* in English. If you were told that a sentence contains the word *the*, based on this word alone there's a pretty good chance that a sentence is written in English. Some words cross language boundaries. The word *casa* is *house* in Spanish, but *casa* is also *house* in Portuguese, making *casa* a term that might help you to narrow down a sentence to a subset of languages, but it won't determine the language for you automatically.

Likelihood is denoted by the following:

$$P(B \mid A)$$

In other words, we are asking, What is the probability of B given that we already know A is true? We could rephrase this as, What is the probability that a sentence contains the word *casa* given that we know it's written in Spanish? and What is the probability that a sentence contains the word *casa* given that we know it's written in Portuguese?

Evidence

Evidence asks for the probability of our known features independent of any unknown features. We will be dividing our likelihood property by the evidence, thus creating a proportional measurement of how the probability of a feature given the known class relates to the probability of the feature as a whole.

Evidence is denoted in the following:

$$P(B)$$

Putting the parts of the Bayes theorem together

Putting it all together, we get the classical Bayes theorem:

$$P(A|B) = \frac{P(A) \times P(B|A)}{P(B)}$$

The Bayes theorem will tell us the probability of *A* (the unknown language class of a tweet) given that we know *B* (a single word) is in a sentence. This needs to be generalized further. We know that there are multiple classes of languages (English, Spanish, Portuguese, and so on.) and each tweet will have multiple words (also known as features).

Since the *evidence* portion of the Bayes theorem involves the same investigation as the likelihood portion, the evidence portion is often ignored when there are multiple features. When dealing with multiple features, we multiply the likelihood of each feature given a class times the prior probability of that class. This can be denoted by the following:

$$P(A \mid B_1, B_2, \ldots, B_n) = P(A)P(B_1 \mid A)P(B_2 \mid A)\ldots P(B_n \mid A) = P(A)\prod_{i=1}^{n} P(B_i \mid A)$$

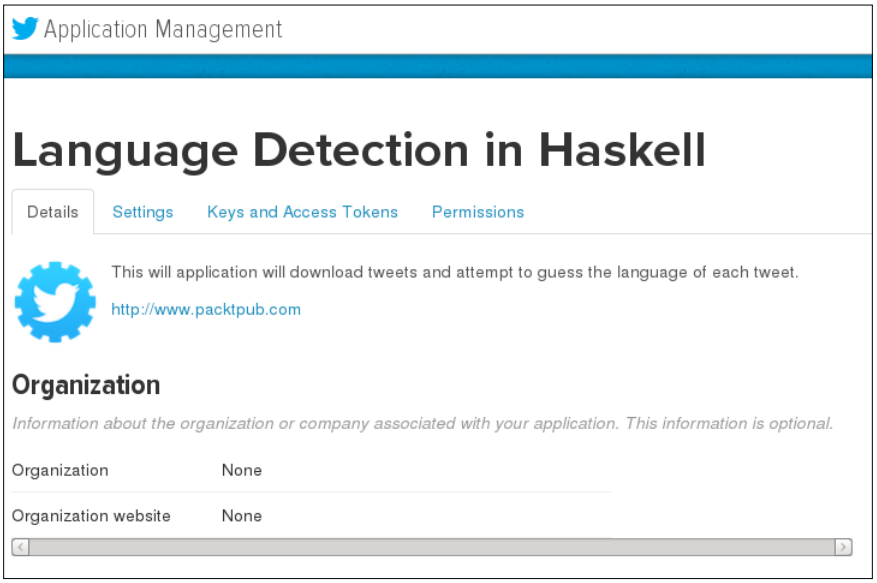
This represents the probability that a feature vector (a list of words represented by B_1 to B_n) represents a class (language *A*) based on our prior knowledge of multiple known features. We perform this calculation for each of our classes and select the class with the highest probability.

But before we can do any of this, we need data.

Creating a Twitter application

We need to collect data from Twitter via Twitter's available APIs. In order to follow along with the book, you are going to need a Twitter account and a Twitter application project. Go to `www.twitter.com` and create an account if you don't have one already.

Sign in to your account and go to `https://apps.twitter.com/` and create a new Twitter application. Here's a screenshot of the application that I used for this chapter:



Upon creating an application, Twitter will generate a **Consumer key** and a **Consumer secret** key for you. It's considered a best practice to keep this information a secret since Twitter will hold the holder of these keys responsible for the activity generated by them. In addition to these two keys, you will have to manually generate an **Access token** and an **Access token secret** in order to access the REST APIs. These keys can be generated from the **Keys and Access Tokens** tab within the Twitter **Application Management** page.

Communicating with Twitter

Now we need to craft some Haskell code that will communicate with the Twitter API and download tweets. The Twitter API uses **OAuth** in order to provide some security for their application. All responses to the Twitter API are returned as JSON objects. The code presented in this chapter for communicating with the Twitter API was adapted from a tutorial from the *FP Complete* website. You can find the full tutorial here: `https://www.fpcomplete.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json`.

Our goal is to download tweets and we aren't picky about which tweets are downloaded. We just need data and plenty of it. The closest API call within the documentation is the search API, which requires that we provide a term on which to search. For this, we selected the search term `a`, which has usage across multiple languages (which is our goal in this exercise). At the time of writing, Twitter allows us to download a maximum of 100 tweets per query and allows us to query the API up to 180 times every 15 minutes (for a total of 18,000 tweets). Complete information about the search API command that we will be using in this chapter can be found here: `https://dev.twitter.com/rest/reference/get/search/tweets`.

The JSON object returned by the Twitter API contains lots of information on each tweet, including the tweet itself, the member who wrote the tweet, the time, and the detected language. Twitter admits that *Language detection is best-effort* in their documentation. Understanding the potential that Twitter might be wrong, we will be using Twitter's assumption of the detected language as the training data for our classifier.

From the GHCi prompt, we import the following libraries:

```
> import Data.HashMap.Strict as HM
> import Data.List as L
```

We should display all of the libraries used in this chapter (and we depend on several). This is the beginning of this chapter's module (`LearningDataAnalysis07`):

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}
module LearningDataAnalysis07 where
import Data.List as L
import Data.Hashable
import Data.HashMap.Strict as HM
import Database.HDBC.Sqlite3
import Database.HDBC
import Control.Concurrent
import Data.Char
import Network.HTTP.Conduit
import Web.Authenticate.OAuth
import Data.Aeson
import GHC.Generics
```

As you can see in these `import` snippets, I've started to use the `as` keyword to differentiate the `Data.List` and the `Data.HashMap.Strict` libraries. Both of these libraries have a function called `map`, and this requires that the two libraries be named.

Let's begin by creating our credentials for the API. Replacing the fields that begin with `YOUR...` with your API credentials, `myoauth` and `mycred`, will allow us to be properly identified by Twitter:

```
myoauth :: OAuth
myoauth =
  newOAuth { oauthServerName     = "api.twitter.com"
           , oauthConsumerKey    = "YOUR CONSUMER KEY"
           , oauthConsumerSecret = "YOUR CONSUMER SECRET KEY"
           }
mycred :: Credential
mycred = newCredential "YOUR ACCESS TOKEN"
                       "YOUR ACCESS TOKEN SECRET"
```

Next, we have to create some data objects that will be pattern-matched with the JSON object returned by the API. Each JSON tweet object returned by the search command will contain various fields. We can select just the fields we desire here.

Within the `User` object, there is the `screenName` field. We collect it here. While we aren't using the `screenName` information in this chapter, we do use it in the next chapter:

```
data User =
    User { screenName :: !String } deriving (Show, Generic)
```

The `Status` object contains the tweet (called `text`), the detected language (called `lang`), and a `User` object as shown in the following function:

```
data Status =
  Status { text :: !String,
           lang :: !String,
           user :: !User } deriving (Show, Generic)
```

Each `Status` object is contained in a list of objects named `statuses` as shown in the following function:

```
data Search =
  Search { statuses :: ![Status] } deriving (Show, Generic)
```

Once our data objects have been defined, we make sure that Haskell recognizes that we will be performing JSON pattern matching on these objects using the following `instance` statements:

```
instance FromJSON User
instance ToJSON User
instance FromJSON Status
instance ToJSON Status
instance FromJSON Search
instance ToJSON Search
```

We then call the `Twitter_Search` function and download tweets. This function will call Twitter's search API using a signed OAuth request and parse what is returned using the `Search` object.

```
twitterSearch :: String -> IO (Either String Search)
twitterSearch term = do
  req <- parseUrl $
  "https://api.twitter.com/1.1/search/tweets.json?count=100&q=" ++
  term
  res <- withManager $ \m -> do
          signedreq <- signOAuth myoauth mycred req
          httpLbs signedreq m
  return $ eitherDecode $ responseBody res
```

This function will return either an error string or a `Search` object.

Creating a database to collect tweets

Now that we have a function to search Twitter, we need to create a database to collect tweets. We will create a `tweets.sql` file in the same manner that we used in *Chapter 2*, *Getting Our Feet Wet*:

```
createTweetsDatabase :: IO()
createTweetsDatabase = do
    conn <- connectSqlite3 "tweets.sql"
    run conn createStatement []
    commit conn
    disconnect conn
    putStrLn "Successfully created database."
  where
    createStatement =
        "CREATE TABLE tweets (message TEXT, user TEXT, language
        TEXT)"
```

Next, we need a separate function for inserting tweets into this database. Again, we will be using a similar technique to insert records that we used in *Chapter 2*, *Getting Our Feet Wet* for CSV records:

```
insertTweetsInDatabase :: [Tweet] -> IO()
insertTweetsInDatabase tweets = do
    conn <- connectSqlite3 "tweets.sql"
    stmt <- prepare conn insertStatement
    executeMany stmt sqlRecords
    commit conn
    disconnect conn
    putStrLn "Successfully inserted Tweets to database."
  where
    insertStatement = "INSERT INTO tweets VALUES (?, ?, ?)"
    sqlRecords = L.map (\(Tweet message language (User user)) ->
                 [toSql message, toSql user, toSql language])
                 tweets
```

Next, we need a function to call our `twitterSearch` function and insert the returned objects into the database via `insertTweetsInDatabase`. We are using `threadDelay` in order to have a delay of five seconds after each API call in order to allow a little breathing time between each call, as seen in the following function:

```
collectTweetsIntoDatabase :: IO()
collectTweetsIntoDatabase = do
    status <- twitterSearch "a"
    either
       putStrLn
       (\(Search statuses) -> insertTweetsInDatabase statuses)
       status
    threadDelay 5000
```

Finally, we collect the tweets. Simply create the database and call `collectTweetsIntoDatabase` 180 times. If written correctly, this should print the success message 180 times on the screen and the database will be populated with 18,000 tweets. It's enough to get us started.

```
> :l LearningDataAnalysis02 LearningDataAnalysis04 LearningDataAnalysis07
> :m LearningDataAnalysis02 LearningDataAnalysis04 LearningDataAnalysis07
> createTweetsDatabase
> mapM_ (\x -> collectTweetsIntoDatabase) [1..180]
```

From the command line, we can pull our tweets from the database.

```
> sqlTweets <- queryDatabase "tweets.sql" "SELECT message, language FROM
tweets"
> let tweets = zip (readStringColumn sqlTweets 0) (readStringColumn
sqlTweets 1)
```

A frequency study of tweets

A frequency function is one that counts the number of times each element is seen in a list. We will be using our frequency function in order to create a unique set of tweets, words, and languages in our database. The function that we will be creating returns a `HashMap` structure and will be used extensively in this chapter. Make sure that you install the library using `cabal`:

```
$ cabal install hashmap
```

This recursive function indexes through each element in a list and creates a mapped value of 1 for the elements that do not currently exist in the `HashMap` and adds 1 to the elements that do exist.

```
frequency :: (Eq k, Data.Hashable.Hashable k, Integral v) =>
             [k] -> HashMap k v
frequency [] = HM.empty
frequency (x:xs) = HM.insertWith (+) x 1 (frequency xs)
```

You can quickly test to see if the function is working with a little help from Dr. Seuss. The title of `one fish two fish red fish blue fish` has five unique words and the word `fish` is repeated four times.

```
> frequency $ words "one fish two fish red fish blue fish"
fromList [("blue",1),("one",1),("two",1),("red",1),("fish",4)]
```

We can pass our listing of tweets to the `frequency` function in order to create a unique list:

```
> let freqTable = frequency tweets
> let uniqueTweets = HM.keys freqTable
> HM.size freqTable
15656
```

It seems that almost 87 percent of the tweets from your author's downloaded dataset represented unique content. We will be using these unique tweets for the remaining phases.

Cleaning our tweets

Now that we have data, we need to scrub this data. We wish to focus on the individual words in a tweet. Our ideal tweet is one that is written in all lowercase letters without any punctuation, without any hashtags, without any links, and without any replies to other users. This mythical tweet is rare and we must adapt the existing tweets to this form.

To scrub our data, I'm going to borrow a function found in the *Haskell Data Analysis Cookbook, Nishant Shukla, Packt Publishing* (I was a technical reviewer for this book; it's an excellent book and I referred back to it regularly when preparing for this book):

```
-- Removes @ replies, hashtags, and links from strings.
clean :: String -> String
clean myString = unwords $ L.filter
    (\myWord -> not (or
                [ isInfixOf "@" myWord
                , isInfixOf "#" myWord
                , isInfixOf "http://" myWord ]))
    (words myString)
```

Now what this function won't do is convert tweets to all lowercase letters and remove punctuation. For that, I created a second function called `removePunctionation`:

```
removePunctuation :: String -> String
removePunctuation myString =
                  [toLower c | c <- myString, or [isAlpha c,
                  isSpace c]]
```

With these two functions in place, we can clean our data to our ideal working conditions:

```
> let cleanedTweets = zip (L.map (removePunctuation.clean.fst)
uniqueTweets) (L.map snd uniqueTweets)
```

Creating our feature vectors

To begin, let's create the frequency table of our languages seen in the database:

```
> let languageFrequency = (frequency . L.map snd) cleanedTweets
> languageFrequency
```

```
fromList [("uk",3),("pt",1800),("th",1),("sl",13),("in",34),("ja",274),("
tl",13),("ar",180),("hi",1),("en",8318),("lv",2),("sk",13),("fi",2),("el"
,4),("vi",1),("ht",16),("es",3339),("pl",50),("da",4),("hu",10),("zh",1),
("nl",13),("ko",17),("tr",73),("und",86),("it",235),("sv",8),("fr",1078),
("is",1),("de",20),("bg",2),("fa",3),("ru",38),("et",3)]
```

You can glance through the list and see the language codes. English (`en`) has just over 8,000 tweets. Next in size is Spanish (`es`) with over 3,000 tweets. Third is Portuguese (`pt`) with 1,800 tweets, and fourth is French (`fr`) with a little over 1,000 tweets. By dividing each of these language counts by the sum (15,656), we have our prior estimation required by the Bayes theorem. Since English is represented by 8,318 tweets, the probability that a tweet will be English will be about 53 percent without knowing the contents of that tweet. (My original search term of `"a"` causes the database to be heavily slanted towards English. For the purposes of this tutorial, that's okay.)

While we're looking at the `languageFrequency` table, let's grab the unique languages represented:

```
> let allLanguages = HM.keys languageFrequency
> length allLanguages
34
```

Next, we need to know the `frequency` table of each word across all languages. This will be similar to our last command, which computed the `frequency` table of languages.

```
> let wordFrequency = (frequency . concatMap words) (L.map fst
cleanedTweets)
> size wordFrequency
34250
```

Our database contains 34,000 unique words across 15,000 unique tweets and over 30 unique languages, all from waiting 15 minutes to download tweets. Not bad at all.

Here's where it gets tricky. We now need a frequency table of each word, frequency with respect to each language. This requires a **HashMap** object of languages that maps to a HashMap object of words and frequencies:

```
> let wordFrequencyByLanguage = (HM.fromList . L.map (\language ->
(language, (frequency . concatMap words . L.map fst) (L.filter (\tweet ->
language == (snd tweet)) cleanedTweets)))) allLanguages
```

We've got HashMap objects embedded within a HashMap. On the first layer of our HashMap object is each two-letter language code. Each two-letter language code maps to another HashMap with the word frequency for words among this language.

At the end of this section, we should have four variables: `allLanguages` (a list of String objects representing each language code), `languageFrequency` (a HashMap of String objects that map to Integer objects), `wordFrequency` (a HashMap of String objects that map to Integer objects), and `wordFrequencyByLanguage` (a HashMap of String objects that map to a HashMap of String objects that map to Integer objects). We will be using these variables when writing our Naive Bayes classifier.

Writing the code for the Bayes theorem

To begin, we will be writing the code for the simple single-feature Bayes theorem:

```
probLanguageGivenWord ::
    String
    -> String
    -> HashMap String Integer
    -> HashMap String Integer
    -> HashMap String (HashMap String Integer)
    -> Double
probLanguageGivenWord
    language
    wordlanguageFrequency
    wordFrequency
    wordFrequencyByLanguage =
    pLanguage * pWordGivenLanguage / pWord
  where
      countTweets = fromIntegral . sum $ elems languageFrequency

      countAllWords = fromIntegral . sum $ elems wordFrequency

      countLanguage = fromIntegral $
                      lookupDefault 0 language languageFrequency

      countWordsUsedInLanguage = fromIntegral . sum . elems $
                                 wordFrequencyByLanguage !
                                 language

      countWord = fromIntegral $ lookupDefault 0 word
                  wordFrequency
```

```
    countWordInLanguage = fromIntegral $ lookupDefault 0 word
                          (wordFrequencyByLanguage ! language)

    pLanguage = countLanguage / countTweets

    pWordGivenLanguage = countWordInLanguage /
                           countWordsUsedInLanguage

    pWord = countWord / countAllWords
```

This code is piecing together each of the individual parts of the Bayes theorem. Under the function's `where` clause, I've created several variables that begin with `count`. These `count` variables tally various important values from the four variables collected in the previous section. The three variables essential to the Bayes theorem are `probLanguage`, `probWordGivenLanguage`, and `probWord`. You should be able to identify how these variables are being calculated in the preceding code.

Let's test this algorithm on the word `house` with the top four languages found in our database:

```
> probLanguageGivenWord "en" "house" languageFrequency wordFrequency
wordFrequencyByLanguage
0.9637796833052451


> probLanguageGivenWord "es" "house" languageFrequency wordFrequency
wordFrequencyByLanguage
0.0


> probLanguageGivenWord "pt" "house" languageFrequency wordFrequency
wordFrequencyByLanguage
0.0


> probLanguageGivenWord "fr" "house" languageFrequency wordFrequency
wordFrequencyByLanguage
0.0
```

As you can see, `house` is a very English word. None of the other three languages return a score greater than 0. Let's look at the word `casa` with the top four languages in our database:

```
> probLanguageGivenWord "en" "casa" languageFrequency wordFrequency
wordFrequencyByLanguage
7.899833469715125e-3
```

```
> probLanguageGivenWord "es" "casa" languageFrequency wordFrequency
wordFrequencyByLanguage
0.4144443354127799

> probLanguageGivenWord "pt" "casa" languageFrequency wordFrequency
wordFrequencyByLanguage
0.5225466167002369

> probLanguageGivenWord "fr" "casa" languageFrequency wordFrequency
wordFrequencyByLanguage
2.3998008047484393e-2
```

The term `casa` appears in all four languages to some degree, but especially in Spanish and Portuguese. There appears to be a slight edge to Portuguese over Spanish in terms of the *dominant ownership* of the word.

Creating a Naive Bayes classifier with multiple features

As we stated earlier in the chapter, of the three parts of the Bayes Theorem, the two more important ones are the prior probability and the likelihood probability. I've extracted out the probability of a language into its own function here. We compute the total number of tweets of this language divided by the total number of tweets:

```
probLanguage :: String
    -> HashMap String Integer
    -> Double
probLanguage language languageFrequency =
    countLanguage / countTweets
  where
    countTweets = fromIntegral . sum $ elems languageFrequency
    countLanguage = fromIntegral $
                    lookupDefault 0 language languageFrequency
```

Next, we find the probability of a word given a language, in which we divide the number of times a word is seen in a language by the count of all occurrences of all words within a language.

The `lookupDefault` function uses a default value of `0`. This means that if `lookupDefault` cannot find a word within a language's HashMap, it returns `0`.

Recall that our formula for computing the probability of a classifier given a set of features requires that we multiply the probability of each feature. The product of any value multiplied by 0 is 0, thus, if a word cannot be identified within a language, our approach will automatically assume that the probability of a match is 0, when it could just mean that we don't have enough data.

This can be done using the `probWordGivenLanguage` function as follows:

```
probWordGivenLanguage :: String
    -> String
    ->HashMap String (HashMap String Integer)
    -> Double
probWordGivenLanguage word language wordFrequencyByLanguage =
    countWordInLanguage / countWordsUsedInLanguage
  where
    countWordInLanguage = fromIntegral .
                          lookupDefault 0 word $
                          wordFrequencyByLanguage ! language
    countWordsUsedInLanguage = fromIntegral . sum . elems $
                               wordFrequencyByLanguage !
                               language
```

Finally, we can craft our Naive Bayes calculation based on multiple features. This function will multiply the probabilities of each word assuming a particular language by the probability of the language itself:

```
probLanguageGivenMessage :: String
    -> String
    -> HashMap String Integer
    -> HashMap String (HashMap String Integer)
    -> Double
probLanguageGivenMessage language message languageFrequency
wordFrequencyByLanguage =
    probLanguage language languageFrequency *
    product (L.map
            (\word ->
            probWordGivenLanguage word language
            wordFrequencyByLanguage)
            (words message))
```

Great! We should test this function on various phrases and languages:

```
> probLanguageGivenMessage "en" "my house is your house"
languageFrequency wordFrequencyByLanguage
1.4151926487738795e-14


> probLanguageGivenMessage "es" "my house is your house"
languageFrequency wordFrequencyByLanguage
0.0


> probLanguageGivenMessage "en" "mi casa su casa" languageFrequency
wordFrequencyByLanguage
2.087214738575832e-20


> probLanguageGivenMessage "es" "mi casa su casa" languageFrequency
wordFrequencyByLanguage
6.3795321947397925e-12
```

Note that the results are tiny numbers. This can be expected when multiplying the probabilities of words. We are not concerned with the size of the probability but with how the probabilities relate to each other across languages. Here, we see that `my house is your house` returns `1.4e-14` in English, which is small, but returns `0.0` in Spanish. We would select the English class for this sentence. We also see `mi casa su casa` returns `6.4e-12` in Spanish and `2.1e-20` in English. Here, Spanish is the selected class.

We can aggregate this process by mapping the function across all known languages:

```
languageClassifierGivenMessage ::
    String
    -> (HashMap String Integer)
    -> (HashMap String (HashMap String Integer))
    -> [(String, Double)]
languageClassifierGivenMessage
    message languageFrequency wordFrequencyByLanguage =
    L.map (\language->
          (language,
           probLanguageGivenMessage
              language message languageFrequency
              wordFrequencyByLanguage))
          (keys languageFrequency)
```

Here, we can take the results of our language classifier and return the maximum (which should be the most likely language):

```
maxClassifier :: [(String, Double)] -> (String, Double)
maxClassifier = L.maximumBy (comparing snd)
```

Testing our classifier

We are finally at a point where we can perform some testing on our approach. We will begin with simple phrases in English and Spanish, since these two languages make up most of the database.

First, we will test five phrases in English:

```
> maxClassifier $ languageClassifierGivenMessage "the quick brown fox
jumps over the lazy dog" languageFrequency wordFrequencyByLanguage
("en",1.0384385163880495e-32)

> maxClassifier $ languageClassifierGivenMessage "hi how are you doing
today" languageFrequency wordFrequencyByLanguage
("en",4.3296809098896647e-17)

> maxClassifier $ languageClassifierGivenMessage "it is a beautiful day
outside" languageFrequency wordFrequencyByLanguage
("en",4.604145482001343e-16)

> maxClassifier $ languageClassifierGivenMessage "would you like to join
me for lunch" languageFrequency wordFrequencyByLanguage
("en",6.91160501990044e-21)

> maxClassifier $ languageClassifierGivenMessage "my teacher gave me too
much homework" languageFrequency wordFrequencyByLanguage
("en",6.532933008201886e-23)
```

Next, we evaluate five phrases in Spanish using the same training data. As I mentioned earlier in the chapter, I don't know Spanish. These phrases were pulled from a Spanish language education website:

```
> maxClassifier $ languageClassifierGivenMessage "estoy bien gracias"
languageFrequency wordFrequencyByLanguage
("es",3.5494939242101163e-10)

> maxClassifier $ languageClassifierGivenMessage "vaya ud derecho"
languageFrequency wordFrequencyByLanguage
("es",7.86551836549381e-13)

> maxClassifier $ languageClassifierGivenMessage "eres muy amable"
languageFrequency wordFrequencyByLanguage
("es",2.725124039761997e-12)

> maxClassifier $ languageClassifierGivenMessage "le gusta a usted aquí"
languageFrequency wordFrequencyByLanguage
("es",6.631704901901517e-15)

> maxClassifier $ languageClassifierGivenMessage "feliz cumpleaños"
languageFrequency wordFrequencyByLanguage
("es",2.4923205860794728e-8)
```

Finally, I decided to throw some French phrases at the classifier again, these phrases were pulled from a French language education website:

```
> maxClassifier $ languageClassifierGivenMessage "cest une bonne idée"
languageFrequency wordFrequencyByLanguage
("fr",2.5206114495244297e-13)

> maxClassifier $ languageClassifierGivenMessage "il est très beau"
languageFrequency wordFrequencyByLanguage
("fr",8.027963170060149e-13)
```

Hopefully, you can see that our classifier, which we built on a small database of 18,000 tweets, was enough to detect the correct language of simple phrases.

Summary

Naive Bayes is a simple theorem that can produce a robust classifier. With a little bit of simple math, we have analyzed the frequency of words used across several thousands of tweets and used that analysis in the creation of a language classifier. With a larger database, we could handle more complex phrases. We also learned how to pull data from Twitter's REST API and learned the powerful features of the HashMap library.

The next chapter looks at another tool in the data analyst's toolkit – **Principal Component Analysis** (**PCA**). The math behind PCA has been used to produce recommendation engines for websites such as Amazon and Netflix. We will continue to use our Twitter database in the next chapter to create a recommendation engine of our own.

8
Building a Recommendation Engine

In the last chapter, we looked at ways of classifying data into categories. We took a difficult problem of language detection and used a Naive Bayesian approach to solve it. We also took a lot of unorganized text data, cleaned it, and converted it into a numerical form. The accomplishing of our goal didn't require a lot of code either, all thanks to Haskell's expressive syntax for recursion and list processing. Haskell's beauty comes from its emphasis on type correctness and functional paradigms.

In this chapter, we will cover the following:

- Analyzing the frequency of words in tweets
- Removing stop words from tweets
- Creating multivariate datasets
- Understanding eigenvalues and eigenvectors
- Performing simple linear algebra in Haskell
- Creating a covariance matrix in Haskell
- Eigen-solving in Haskell
- Creating a recommendation engine based on PCA

The problems that we have encountered so far looked at data that fit nicely into one or a maximum of two features. In *Chapter 4*, *Plotting*, we looked at the share prices of various companies over time. In *Chapter 5*, *Hypothesis Testing*, we compared the runs per game for matches that were played at home stadiums to the runs per game for matches that were played at away stadiums in baseball data. In *Chapter 6*, *Correlation and Regression Analysis*, we compared the runs per game to the win percentage in baseball data. The data analyst's job is relatively simple when working with bivariate data. In each example, the primary task is to compare the independent (input) variable to the dependent (output) variable. Life would be so much easier if we could reduce all our problems to a single input variable and an output variable. We alluded to this in the chapter on the estimation of the win percentage of baseball teams—there is more to winning a game than the scoring of runs. Some variables help us to explain the output (such as the runs scored per game and the on-base percentage). However, we won't expect some variables to do the same (such as the temperature at the starting time of the game). You should not immediately rule out these obscure variables, but you can fall into the trap of trying to look for correlations in everything. At some point, you need to make a decision about which variables to include and ignore.

In *Chapter 6*, *Correlation and Regression Analysis*, we discussed the statistical measure of covariance. Covariance measures how two variables relate to each other. A positive covariance score indicates that the two variables are related. A negative score indicates that the two variables are inversely related. A covariance score that is close to 0 (the score can be either positive or negative) indicates that the two variables are probably not related. We also discussed the Pearson's r^2 correlation coefficient, which is a normalized squared version of the covariance score. The Pearson's r^2 score allows us to quickly judge the strength at which two variables correlate. A Pearson's r^2 score of 1 indicates that the two variables correlate and a score of 0 indicates that the two variables do not correlate. When working with lots of variables, our goal should be to improve the results by selecting a subset of input variables with the largest variance scores with the output variable.

In this chapter, we will look at recommendation engines. Websites such as Amazon will recommend products for you to purchase based on your prior purchases. Netflix famously offered 1 million dollars to a team that could design a system to improve their already excellent movie recommendation engine. Facebook will recommend people as your friends and Twitter will recommend people who might have interests that are similar to yours. There are plenty of variables that go into a recommendation engine, but what should come out is a single list of items with a score associated with each item. A select number of top-ranked items are passed along to you as your top-ranked recommendations. Some of these input variables are directly relevant to our recommendations and the others aren't.

Our goal in this chapter is to build our own recommendation engine based on the Twitter data that was obtained using the methods described in *Chapter 7*, *Naive Bayes Classification of Twitter Data*. After writing *Chapter 7*, *Naive Bayes Classification of Twitter Data*, I have been collecting more tweets. The database used in this chapter consists of just over 50,000 tweets, with a little over half of them representing tweets in the English language. We will mine the database consisting of just the English tweets and develop a recommendation engine to find like-minded users to any other user.

This problem is rather difficult to solve and there are lots of ways to approach this. In this chapter, we will attempt to solve it by discovering pairs of users who share the usage rate of commonly used words. The general thinking in this approach is that if two people tend to use the same words in their speech, they will share even more things in common. (I picked this approach because it allows us to use the same dataset collected in the last chapter.) To accomplish this, we need to discover the most frequently used words that were tagged by Twitter with the `en` language code.

Before we proceed further, we need to discuss the necessary software for this chapter. You need to install LAPACK on your system. LAPACK is a collection of algorithms that are used to solve common matrix-related tasks such as the eigenvalue decomposition problem. On Debian-based systems, you should be able to download it using `apt-get`, as follows:

```
$ sudo apt-get install liblapack-dev
```

You will also need to install the Haskell matrix wrapper modules for LAPACK called `hmatrix` using `cabal`, as follows:

```
$ cabal install hmatrix
```

We will discuss the `Numeric.LinearAlgebra` libraries later on in the chapter. Here are the necessary libraries used in this chapter for the `LearningDataAnalysis08` Haskell module:

```
module LearningDataAnalysis08 where
import Numeric.LinearAlgebra.Data
import Numeric.LinearAlgebra.Algorithms
import Numeric.LinearAlgebra.HMatrix
import Data.List as L
```

Analyzing the frequency of words in tweets

We will begin by pulling English tweets from our database. From the Haskell command line, we will query the database in the following way:

```
> :l LearningDataAnalysis04 LearningDataAnalysis06 LearningDataAnalysis07
LearningDataAnalysis08


> :m LearningDataAnalysis04 LearningDataAnalysis06 LearningDataAnalysis07
LearningDataAnalysis08


> import Data.HashMap.Strict as HM
> import Data.List as L


> tweetsEnglish <- queryDatabase "tweets.sql" "SELECT message, user FROM
tweets WHERE language='en'"
> let tweets = zip (readStringColumn tweetsEnglish 0) (readStringColumn
tweetsEnglish 1)
```

Using the `frequency` function presented in the last chapter, we will compute the set of unique tweets, as follows:

```
> let freqTable = frequency tweets
> -- Number of unique tweets
> HM.size freqTable
27348
> let uniqueTweets = keys freqTable
```

After writing the last chapter, I've collected 27,000 unique English tweets. Just as we did in the last chapter, we must clean our tweets:

```
> let cleanedTweets = L.map (\(message, user) -> (removePunctuation $
clean message, user)) uniqueTweets
```

We can now build a `frequency` HashMap of our cleaned tweets in the following way:

```
> let wordsFrequency = frequency $ concatMap (words . fst) cleanedTweets
```

A note on the importance of removing stop words

Now, we must take our word `frequency` HashMap, convert it into a list, and sort that list from largest to smallest. We will then take the first 25 items from the list:

```
> let topWords = take 25 $ sortBy (\(_, c1) (_, c2) -> compare c2 c1) $
HM.toList wordsFrequency

> topWords

[("a",30028),("rt",12594),("to",9815),("the",7754),("i",7506),("you",7
425),("and",5403),("for",5337),("of",5312),("in",5187),("is",4832),("
on",2906),("it",2723),("me",2665),("my",2648),("with",2617),("be",2564),(
"have",2507),("that",2281),("this",2203),("if",1959),("your",1891),("just
",1832),("at",1807),("like",1732)]
```

Looking through this list, you will find that it's not interesting at all. The top two words are `a` (which was the term used to gather these tweets) and `rt` (which is Twitter speak for retweet). The rest of the list consists of common words that are known as function words. Function words are the words that are ambiguous in meaning until used in a sentence to provide some context. We will filter these words out in order to work on more interesting data. In the field of natural language processing, these words are called **stop words**. With a search using your favorite search engine of choice, you can find websites with lists stop words. I made my own list and added it to my `LearningDataAnalysis08` module. A few words have been added to this list, such as `rt` (for retweet), `mt` (for modified tweet), and `u` (which is short for "you"):

```
stopWords :: [String]
stopWords = ["a", "about", "above", "above", "across", "after",
"afterwards", "again", "against", "all", "almost", "alone", "along",
"already", "also", "although", "always", "am", "among", "amongst",
"amoungst", "amount",  "an", "and", "another", "any", "anyhow",
"anyone", "anything", "anyway", "anywhere", "are", "around", "as",
"at", "back", "be", "became", "because", "become", "becomes",
"becoming", "been", "before", "beforehand", "behind", "being",
"below", "beside", "besides", "between", "beyond", "bill", "both",
"bottom", "but", "by", "call", "can", "cannot", "cant", "co", "con",
"could", "couldnt", "cry", "de", "describe", "detail", "do", "done",
"dont", "down", "due", "during", "each", "eg", "eight", "either",
"eleven", "else", "elsewhere", "empty", "enough", "etc", "even",
"ever", "every", "everyone", "everything", "everywhere", "except",
"few", "fifteen", "fifty", "fill", "find", "fire", "first", "five",
"for", "former", "formerly", "forty", "found", "four", "from",
"front", "full", "further", "get", "give", "go", "got", "had", "has",
"hasnt", "have", "he", "hence", "her", "here", "hereafter", "hereby",
"herein", "hereupon", "hers", "herself", "him", "himself", "his",
```

```
"how", "however", "hundred", "i", "ie", "if", "im", "in", "inc",
"indeed", "interest", "into", "is", "it", "its", "itself", "just",
"keep", "last", "latter", "latterly", "least", "less", "ltd", "made",
"many", "may", "me", "meanwhile", "might", "mill", "mine", "more",
"moreover", "most", "mostly", "move", "much", "must", "my", "myself",
"name", "namely", "neither", "need", "never", "nevertheless",
"next", "nine", "no", "nobody", "none", "noone", "nor", "not",
"nothing", "now", "nowhere", "of", "off", "often", "on", "once",
"one", "only", "onto", "or", "other", "others", "otherwise", "our",
"ours", "ourselves", "out", "over", "own", "part", "per", "perhaps",
"please", "put", "rather", "re", "same", "see", "seem", "seemed",
"seeming", "seems", "serious", "several", "she", "should", "show",
"side", "since", "sincere", "six", "sixty", "so", "some", "somehow",
"someone", "something", "sometime", "sometimes", "somewhere", "still",
"such", "system", "take", "ten", "than", "that", "the", "their",
"them", "themselves", "then", "thence", "there", "thereafter",
"thereby", "therefore", "therein", "thereupon", "these", "they",
"thick", "thin", "third", "this", "those", "though", "three",
"through", "throughout", "thru", "thus", "to", "together", "too",
"top", "toward", "towards", "twelve", "twenty", "two", "un", "under",
"until", "up", "upon", "us", "very", "via", "want", "was", "we",
"well", "were", "what", "whatever", "when", "whence", "whenever",
"where", "whereafter", "whereas", "whereby", "wherein", "whereupon",
"wherever", "whether", "which", "while", "whither", "who", "whoever",
"whole", "whom", "whose", "why", "will", "with", "within", "without",
"would", "yet", "you", "your", "youre", "yours", "yourself",
"yourselves", "rt", "mt", "u"]
```

Once your stop list has been created, you can filter out these words in the following way:

```
> let notStopWords = L.filter (\(word, _) -> notElem word stopWords) (HM.
toList wordsFrequency)

> let topWords = take 25 . L.map fst $ sortBy (\(_, c1) (_, c2) ->
compare c2 c1) notStopWords

> topWords

["like", "amp", "day", "good", "new", "love", "time", "follow", "great",
"today", "make", "lot", "people", "video", "know", "life", "happy",
"look", "think", "girl", "win", "photo", "way", "little", "really"]
```

You might argue that there are some words in this list that represent stop words. I hope that you agree that this list contains words that are more interesting than our previous set of stop words. We will use this set of 25 words in order to represent the 25 features of each person represented in our dataset.

Working with multivariate data

With our desired word list created, we must now compile the feature vector for each user in our database. A feature vector is a set of features representing an entity in our dataset. You can think of a set of feature vectors as a spreadsheet. At the top of our spreadsheet is a listing of columns. The first column contains the first feature, the second column contains the second feature, and so forth. There will be a column per feature in the dataset. The number of columns (or features) in a dataset will be represented by the `d` variable (which is short for dimensionality). Our example problem utilizes 25 dimensions for every user. Each row represents one full entity in our dataset and will contain 25 listings. The number of rows (or observations) in a dataset will be represented by the `n` variable (which is used to count items). Our matrix should consist of *n-by-d* values.

We will build a HashMap of a HashMap of `Strings` representing a list of users, their words, and the usage count of each word. This step will be almost identical to the method performed in the previous chapter. We will compile a list of unique users to our dataset and then compile our listing of word frequencies by user:

```
> let userFrequency = frequency $ L.map snd uniqueTweets

> let allUsers = keys userFrequency

> let wordFrequencyByUser = HM.fromList $ L.map (\user -> (user,
frequency $ concatMap words [ message | (message, thisUser) <-
cleanedTweets, thisUser == user ])) allUsers
```

Next, we will create a matrix. This matrix should be `n-by-d` in size and should consist of several thousand rows and 25 columns. Note that I will use the `fromIntegral` function in this step. This is done to ensure that the variables are in the `Double` data type in the subsequent steps:

```
> let getFrequencyOfWordByUser user word = fromIntegral $ (HM.
lookupDefault 0 word (wordFrequencyByUser HM.! user)) :: Double

> let wordsMatrix = L.map (\user -> L.map (\word ->
getFrequencyOfWordByUser user word) topWords) allUsers
```

Describing bivariate and multivariate data

Let's take a break from our **wordsMatrix** function and talk about bivariate data again. In particular, I want to return to the discussion of the problem presented in *Chapter 6*, *Correlation and Regression Analysis*, where we compared the runs per game to the win percentage in baseball data. In that chapter, we used regression analysis to create a linear formula to describe how the runs per game could be used to estimate the win percentage of a team. What we would like to do is expand this topic to allow for more than just two variables, and it will require some linear algebra in order to solve our problem.

When trying to find a linear regression of our baseball data, we discussed the concept of covariance and then computed the covariance of each of our variables. In particular, we wanted to maximize the variance between the two variables. There were three possible covariance combinations:

- The runs scored per game and the runs scored per game
- The win percentage and the win percentage
- The runs scored per game and the win percentage

Any time a variable is compared with itself to find the covariance, we are in essence computing the variance. We can arrange the covariance scores into a matrix in the following way:

	1	**2**
1	The variance of the runs scored per game	The covariance of the runs scored per game and the win percentage
2	The covariance of the runs scored per game and the win percentage	The variance of the win percentage

We will describe the features of this covariance matrix. Each row in the matrix represents a variable in our dataset. Each column in the matrix also represents a variable in the same order as the row order. When a column number and a row number are equal, it should hold the variance of a feature of the data. When the column and row in our matrix equals 1, that cell holds the variance of the runs scored per game. When the column and row equals 2, that cell holds the variance of the win percentage. When the indices of the matrix are not equal (either [1,2] or [2,1]) we see the covariance of the runs scored per game and the win percentage.

When expanding this idea to more than two variables, maintain these rules. Now, there are *d* variables (representing the dimensionality of the dataset). Like before, each row and column in the matrix represents a variable in our dataset. The order in which the variables are represented are the same across the rows and columns. When a row and column index are equal, we compute the variance of that variable. When the column and row number are not equal, we compute the covariance of the variable of the column index and the variable of the row index. Since the covariance of the two variables is the same regardless of the order in which they are stated, we should find the same covariance when the two indices are flipped. Hence, our matrix is always symmetrical from the top left of the matrix to the bottom right of the matrix.

To state this in mathematical terms, we have a dataset called *X* consisting of *n* observations and *d* dimensions. A column of values will be identified by X_i. We wish to create a covariance matrix called *C*. Our covariance matrix will be `d-by-d` in size. The equation for variance is denoted as follows:

$$C_{ii} = variance(X_i) = E\left[\left(X_i - E[X_i]\right)^2\right]$$

The equation for covariance is denoted as follows:

$$C_{ij} = covariance(X_i, X_j) = E\left[\left(X_i - E[X_i]\right)\left(X_j - E\left[X_j\right]\right)\right]$$

Eigenvalues and eigenvectors

Eigenvalues and Eigenvectors are used to measure how a matrix is stretched in various directions, and they are perfect if you want to discover the directions in which a dataset is stretched. We will try to solve the following equation to discover the eigenvalues and eigenvectors of a matrix:

$$Av = \lambda v$$

Explaining the preceding formula English, for a square matrix *A* that has *d* rows and *d* columns, we are trying to find the column matrix *v* (which is a matrix of exactly one column and *d* rows) and a value called lambda. **Lambda** is a single number (also called a scalar). When we multiply lambda with the matrix *v*, we should get the same result as the product of matrix *A* and matrix *v*. Lambda is called the *eigenvalue* and the column matrix *v* is called the *eigenvector*.

This problem is known as **eigenvalue decomposition**, and it's how we reduce an entire matrix down to just a vector and a scalar for that vector. There can be multiple ways to satisfy this problem for a given matrix *A*. For our purpose, we will use the covariance matrix *C* as our matrix for eigenvalue decomposition, and we expect to get back *d* eigenvalues and *d* eigenvectors. Eigenvalues and eigenvectors represent pairs—the first eigenvalue is paired with the first eigenvector column, the second eigenvalue is paired with the second eigenvector column, and so forth. Because of this pairing nature, it is important to maintain the order of the eigenvalues and eigenvectors. Should the order of the eigenvalues change, the eigenvectors need to be reordered accordingly. (The `hmatrix` library does not require us to reorder eigenvalues, so this should not be an issue).

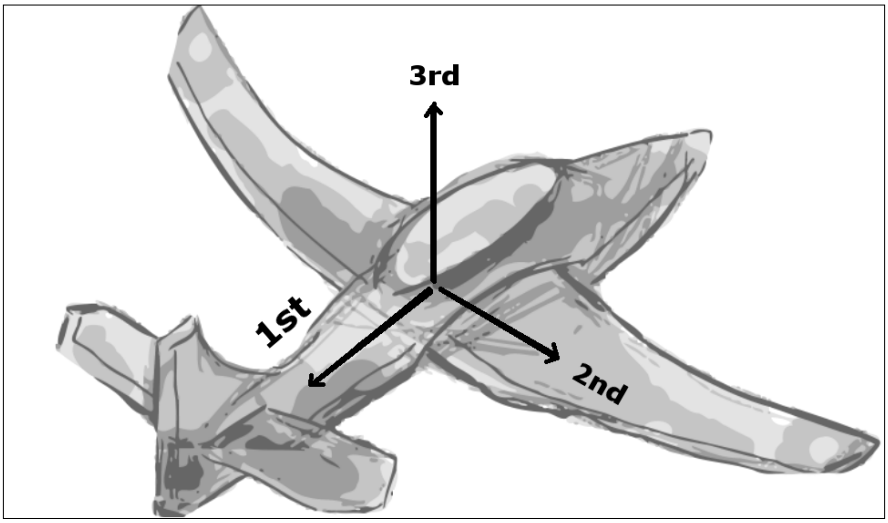
An interesting feature of the eigenvectors returned by the process of eigenvalue decomposition is that each vector is orthogonal to every other vector. This means that the angle between any two eigenvectors will be π/2 radians (or 90 degrees if you measure angles in degrees). The value of each eigenvalue is also important. The larger the eigenvalue, the more our data is skewed in the direction of the associated eigenvector and accounts for a major part of the data's variance.

The airplane analogy

The analogy that I use to understand eigenvectors consists of thinking of an **airplane**. Our dataset will be the points scattered along the surface of the airplane. The airplane has a massive body, which accounts for most of the data. Also, there are two thin wings which stretch out from the either side of the body, and some of the data will be scattered along the wings. The airplane is tall, but the length of the airplane and the wingspan are greater than the height. We typically don't think of airplanes as being tall, but they are still tall objects. If we were to take our set of points in three dimensions, we can compute the covariance matrix of this dataset and produce a 3-by-3 matrix. We can pass this covariance matrix to the eigenvalue decomposition process.

An airplane is a three-dimensional object. So, the results of our eigenvalue decomposition process will have 3 eigenvalues and 3 eigenvectors. We will look at the eigenvalues and rank them in order from the largest to the smallest, making sure to remember their order since they each have an associated eigenvector. We will examine the largest eigenvalue and its associated eigenvector. When we position this eigenvector at the center of the airplane, note that it will extend below the body of the airplane (towards either the nose of the airplane or the tail of the airplane).

Most of our data is in the body of the airplane and the decomposition process recognizes this at the start. When we take a look at the eigenvector associated with the second largest eigenvalue and position it at the center of the airplane, the eigenvector points towards one of the two wings. Some of our data was spread along the wings, and the eigenvalue decomposition makes this the second ranked feature of the data. Also, this vector will be orthogonal to the vector pointing down the body of the airplane. Finally, we will examine the eigenvector for the smallest eigenvalue and position it at the center of the airplane. It will point either directly above or below the airplane along the third dimension. The smallest eigenvalue accounts for the least amount of variance in our data. Also, the third eigenvector is orthogonal to the first two eigenvectors. The following figure shows the three-dimensional structure of an airplane:



Here's what makes the eigenvalue decomposition process so useful—we never mentioned the original orientation of the airplane. It doesn't matter! This process works when the airplane is flat on the ground, in the air making a banking turn, upside down, lifting off, making a descent, or any other conceivable orientation. The eigenvalue decomposition process will first notice the body, then the wings, and finally the height of the airplane, and the vectors will orient themselves to the original orientation of the airplane when the data was first obtained.

Preparing our environment

When performing the eigenvalue decomposition for a small matrix of *2-by-2* or *3-by-3*, there is a simple set of steps that we can take to produce our eigenvalues and eigenvectors. When working with larger matrices, this problem is rather tedious and it's best left to a computer to discover the eigenvalues. A study of the algorithms used to discover eigenvalues goes well beyond the scope of this book. We will use the `hmatrix` library and LAPACK to compute the eigenvalues and eigenvectors of our covariance matrices.

Performing linear algebra in Haskell

Through `hmatrix`, you will have access to a large collection of linear algebra functions. In this section, we are going to provide a brief introduction to this library. If you have ever taken a linear algebra course, you will have learned how powerful matrix operations are.

To begin, let's create a *3-by-4* matrix consisting of values from 1 to 12. This can be done using the `matrix` function in the following way:

```
> let a = matrix 4 [1 .. 12]
> a
(3><4)
 [ 1.0,  2.0,  3.0,  4.0
 , 5.0,  6.0,  7.0,  8.0
 , 9.0, 10.0, 11.0, 12.0 ]
```

We can compute the transpose of this matrix using the `tr` function. Here, we will compute the transpose of *a*, as follows:

```
> tr a
(4><3)
 [ 1.0, 5.0,  9.0
 , 2.0, 6.0, 10.0  , 3.0, 7.0, 11.0
 , 4.0, 8.0, 12.0 ]
```

We can also perform matrix multiplication using the `mul` operator. Here, we will multiply "*a*" with the transpose of *a*, as follows:

```
> mul a $ tr a
(3><3)
 [  30.0,  70.0, 110.0
 ,  70.0, 174.0, 278.0
 , 110.0, 278.0, 446.0 ]
```

This set of modules allows for the use of the standard math operators (+, -, /, *, and ^) for pair-wise operations on matrices:

```
> a + a
(3><4)
 [  2.0,  4.0,  6.0,  8.0
 , 10.0, 12.0, 14.0, 16.0
 , 18.0, 20.0, 22.0, 24.0 ]
```

Computing the covariance matrix of a dataset

The function that is used to compute the covariance matrix of a dataset (`meanCov`, which comes with `hmatrix`) uses a slightly different formula to the one we presented in the chapter on linear regression. Rather than use the expectation function, `meanCov` will sum the necessary values then divide the covariance by *(n-1)* rather than divide by *n*. This is done to signify that this is a sampling of the data (and thus, there is always an unknown quality to the data) rather than a complete sampling of all the data. The equation for this can be denoted as follows:

$$C_{ij} = covariance(X_i, X_j) = \frac{1}{n-1}\sum(X_i - E[X_i])(X_j - E[X_j])$$

To illustrate the use of `meanCov`, we are going to use the baseball dataset from *Chapter 6*, *Correlation and Regression Analysis*. Rather than using linear regression to estimate a line, we are going to use eigenvalue decomposition. We need to get our baseball data into a matrix data structure. I've morphed the current baseball dataset consisting of 30 pairs of runs per game and win percentages into a list of `Double` values.

We've reconstructed the baseball dataset from scratch in the following statements so that you don't have to flip back to that portion of the book:

```
> :l LearningDataAnalysis06

> queryDatabase "winloss.sql" "SELECT COUNT(*) FROM winloss"

> queryDatabase "winloss.sql" "SELECT COUNT(*) FROM winloss WHERE
awayscore == homescore;"


> homeRecord <- queryDatabase "winloss.sql" "SELECT homeTeam,
SUM(homescore > awayscore), SUM(homescore), COUNT(*) FROM winloss GROUP
BY homeTeam;"


> awayRecord <- queryDatabase "winloss.sql" "SELECT awayTeam,
SUM(awayscore > homescore), SUM(awayscore), COUNT(*) FROM winloss GROUP
BY awayTeam;"


> let totalWins = zipWith (+) (readDoubleColumn homeRecord 1)
(readDoubleColumn awayRecord 1)

> let totalRuns = zipWith (+) (readDoubleColumn homeRecord 2)
(readDoubleColumn awayRecord 2)

> let totalGames = zipWith (+) (readDoubleColumn homeRecord 3)
(readDoubleColumn awayRecord 3)

> let winPercentage = zipWith (/) totalWins totalGames

> let runsPerGame = zipWith (/) totalRuns totalGames

> let baseball = L.map (\(a,b) -> [a,b]) $ zip winPercentage runsPerGame


> :t baseball

baseball :: [[Double]]

> baseball

[[0.6049382716049383,4.771604938271605],[0.39751552795031053,3.8136645962
73292],[0.4876543209876543,3.537037037037037],[0.5925925925925926,4.35185
1851851852], … [content clipped]
```

We can now compute the covariance matrix of our dataset by calling `meanCov`. This function will return two values. The first value is a vector consisting of the mean of each column of data. The second value is the covariance matrix:

```
> let (baseballMean, baseballCovMatrix) = meanCov $ fromLists baseball

> baseballMean
```

```
fromList [4.066924826828208,0.4999948879175932]
> baseballCovMatrix
(2><2)
 [   0.1204356428862124,  8.350015780569988e-3
 , 8.350015780569988e-3, 3.4790736953854207e-3 ]
```

Discovering eigenvalues and eigenvectors in Haskell

Following this step, we will perform the eigenvalue decomposition process. The `eigSH` function will return a list of eigenvalues and eigenvectors presorted so that the largest eigenvalues are ordered first:

```
> let (baseballEvalues, baseballEvectors) = eigSH baseballCovMatrix
> baseballEvalues
fromList [0.12102877720686984,2.8859393747279805e-3]
> baseballEvectors
(2><2)
 [   -0.9974865990115773, 7.085537941692915e-2
 , -7.085537941692915e-2,  -0.9974865990115773 ]
```

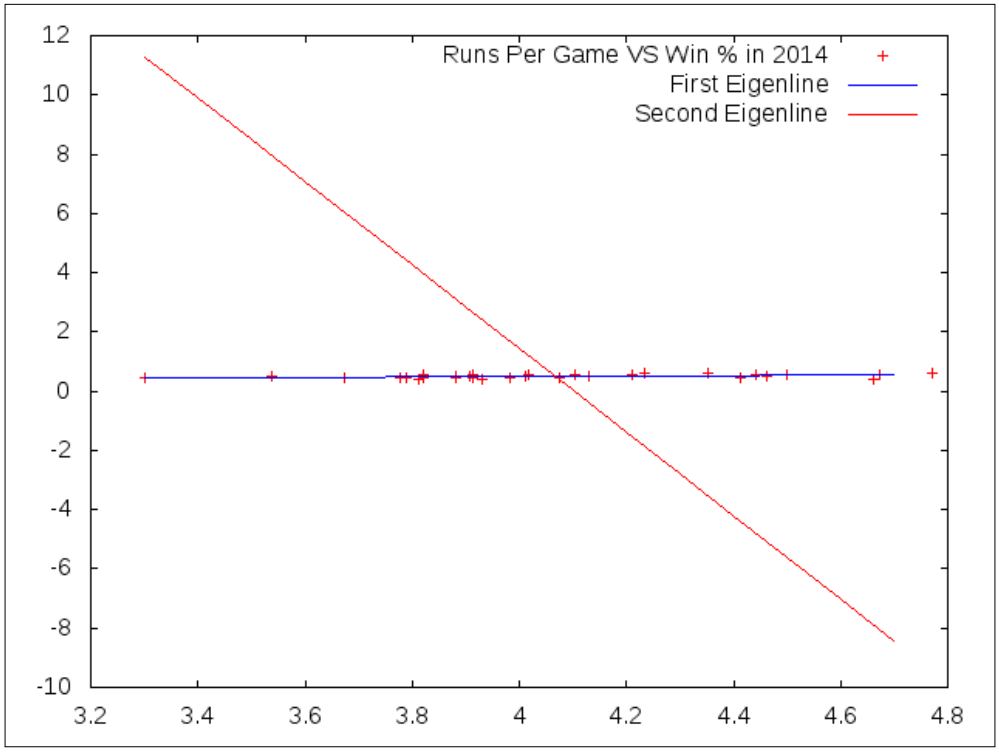
The eigenvectors are returned in a single matrix. Each eigenvector can be identified in each column. The largest eigenvalue (0.12) is associated with the first column of the eigenvector matrix (-0.9975, -7.0855e-2). The second largest eigenvalue (2.89e-3) is associated with the second column of the eigenvector matrix (7.0855e-2, -0.9975).

Here, we will visualize what happens when we plot a line with the first eigenvector of (-0.9975, -7.0855e-2) that is centered at the mean position (4.0667, 0.5) of the dataset with a blue line. We do the same for our second eigenvector (7.0855e-2, -0.9975) with a red line. I call these lines *eigenlines*. This is done using the following statements:

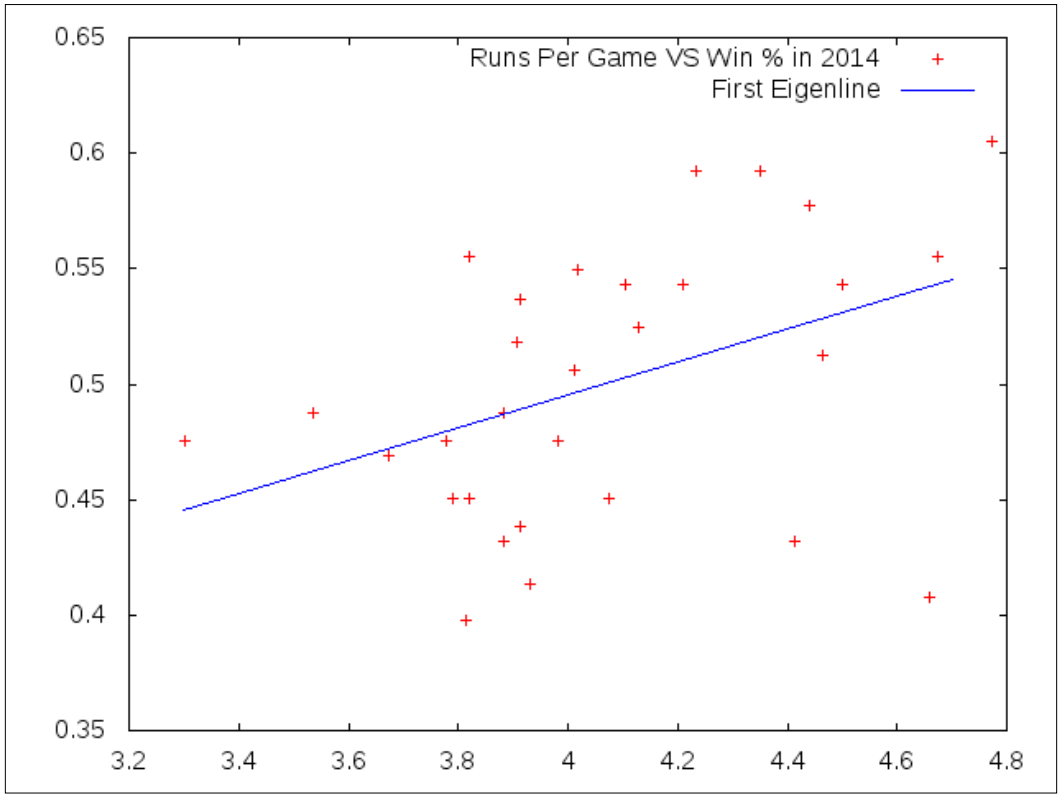
```
> let xmean = 4.0667
> let ymean = 0.5
> let samplePoints = [3.3, 3.4 .. 4.7]

> let firstSlope = (-7.085537941692915e-2) / (-0.9974865990115773)
> let secondSlope = (-0.9974865990115773) / (7.085537941692915e-2)
```

```
> let firstIntercept = ymean - (xmean * firstSlope)
> let secondIntercept = ymean - (xmean * secondSlope)

> let firstEigenline = zip samplePoints $ L.map (\x -> x*firstSlope +
firstIntercept) samplePoints
> let secondEigenline = zip samplePoints $ L.map (\x -> x*secondSlope +
secondIntercept) samplePoints

> import Graphics.EasyPlot
> plot (PNG "runs_and_wins_with_eigenlines.png") [Data2D [Title "Runs Per
Game VS Win % in 2014"] [] (zip runsPerGame winPercentage), Data2D [Title
"First Eigenline", Style Lines, Color Blue] [] firstEigenline, Data2D
[Title "Second Eigenline", Style Lines, Color Red] [] secondEigenline]
True
> plot (PNG "runs_and_wins_first_eigenline.png") [Data2D [Title "Runs Per
Game VS Win % in 2014"] [] (zip runsPerGame winPercentage), Data2D [Title
"First Eigenline", Style Lines, Color Blue] [] firstEigenline]
```

The preceding command would give the following chart as a result:



The scale is causing some distortion in this image, but the red and blue lines are perpendicular to each other. By removing the red line for the second red eigenline from the image, we see an image that is almost identical to the one presented in *Chapter 6*, *Correlation and Regression Analysis*. Any difference between the following image and the similar image depicting linear regression can be chalked up to our differences in computing the covariance:



Principal Component Analysis in Haskell

The previous example demonstrates an important trait of the eigenvalue decomposition process—some of the vectors produced by this process are not useful to us. Sometimes, we just need a few vectors to do the job. In this example, we only needed to use the first eigenvector and we could easily identify this vector because it had the largest associated eigenvalue.

Principal Component Analysis (PCA) is the process by which we compute the eigenvalues of a covariance matrix, rank them from the largest to the smallest, and then keep just the *t* largest eigenvalues and vectors. The `eigSH` function goes through the trouble of eigenvalue discovery and ranking for us. It's our job to select the number of top eigenvalues that we want to keep.

Once we have gathered our top eigenvalues and eigenvectors, we can map our dataset to a lower dimensional space. If *X* is the *n-by-d* matrix representing the original dataset and *T* is the *d-by-t* matrix (where *t* is a value less *d*) representing the top eigenvector matrix, we can perform the following to create an *n-by-t* matrix named *S*. Just a reminder about what each of these letters mean—*n* represents the number of records in our dataset, *d* represents the original dimensionality of the dataset, and *t* represents the desired dimensionality. We are morphing *X* using *T* to create *S*:

$$S = \left(T' \times X'\right)'$$

A bit of an explanation is due with regards to this formula. Each variable in this equation represents a matrix. The multiplication symbol refers to the process of matrix multiplication (performed by calling the `mul` function). The apostrophe character to the right of a matrix refers to the matrix transpose operation (performed by calling the `tr` function).

The *S* matrix is now a dataset with fewer features than our original *X* dataset. Think of this *S* matrix as a dataset that's been cleaned somewhat by dropping some of the dimensions of the data that did not contribute enough variance, which helps us better explain the dataset.

Let's craft our own PCA function in Haskell. This function will take a List of `Double` values representing our original dataset (named records) as well as an `Integer` value representing the desired number of eigenvectors that we wish to use (named `top`). The following method will return a matrix representing our original dataset in a lower dimensional space using only the `top` dimensions instead of all the dimensions:

```
principalComponentAnalysis :: [[Double]] -> Integer -> Matrix
  Double
principalComponentAnalysis records top =
    tr $ mul (tr topOrderedEvectors) (tr featureVectors)
  where
    featureVectors = fromLists records
    (_, covMatrix) = meanCov featureVectors
    (_, evectors) = eigSH covMatrix
    topOrderedEvectors = takeColumns (fromInteger top) evectors
```

Building a recommendation engine

When we left the recommendation engine problem at the beginning of the chapter, we created the `wordsMatrix` variable. We should now have enough background with regards to the benefits of the eigenvalue decomposition process, which will help us to finish the creation of our recommendation engine. We can use the `principalComponentAnalysis` function and produce our own dataset based on `wordsMatrix`, as follows:

```
> let pcaMatrix = principalComponentAnalysis wordsMatrix 5
```

The `pcaMatrix` function is a compressed form of the `wordsMatrix` variable, which focuses on the 5 vectors with the highest variance.

Finding the nearest neighbors

Each row in the `pcaMatrix` function represents one user. If we were to plot this dataset (which would be a challenge in five dimensions), we would see points scattered through the space. These points would be gathered near other points with a high similarity. What we are going to do next is use `pcaMatrix` and seek out the 5 nearest neighbors to a given user at a specified index.

To find the point nearest to another point, you need a distance measure. A commonly used distance measure is known as the Euclidean distance. The equation for this can be denoted as follows:

$$Euclidean(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

This approach returns the true distance between two points, say *a* and *b*, but there is a way to optimize this function. When comparing two distances, we are trying to figure out the distance that is the shortest, but we don't require the distances themselves. We can still get the information desired by dropping the square root function. Comparing Euclidean distances with or without a square root will return the same result (and the square root operation can be a costly calculation).

Because of this, we will use the Euclidean squared distance:

$$EuclideanSquared(a,b) = \sum_{i=1}^{n} (a_i - b_i)^2$$

In order to perform this operation in code, we are going to isolate a single desired row in our matrix, repeat it *n* times (where *n* is the number of rows in the matrix), and then use matrix operations to return the Euclidean squared distance in the following way:

```
euclideanDistanceSqrdFromRow :: Matrix Double) -> Integer ->
  Vector Double
euclideanDistanceSqrdFromRow records row =
    sumOfSquaresVector
  where
    d = cols records
    copyMatrix = repmat
      (subMatrix (fromIntegral row, 0) (1, d) records) (rows
      records) 1
      diffMatrix = records - copyMatrix
      diffSqrdMatrix = diffMatrix * diffMatrix
      sumOfSquaresVector = sum $ toColumns diffSqrdMatrix
```

The result of this operation will be a vector (not a matrix), that is, *n* elements long representing distances from a selected index to every index. The distance from the selected index to itself should be 0.

Next, we need to come up with an ordering for the values returned by this function. Here's what I came up with. This algorithm assumes that the list of distances has been *nubsorted*. This means that the distances have been sorted and all the duplicates have been removed. For each value in the `nubSorted` list, we return a list of all the index positions in the unsorted list that match this position. This is accomplished by using the `elemIndices` function, as follows:

```
order :: Eq a => [a] -> [a] -> [Integer]
order unsorted nubSorted =
  concatMap (\x -> L.map toInteger $ L.elemIndices x unsorted)
  nubSorted
```

Finally, we put it all together into a function that will recommend users to us based on our usage of popular words. This function will take a matrix returned by the `principalComponentAnalysis` function and search the `knn` nearest index positions in index position's row by using the Euclidean similarity, as follows:

```
findClosestFeaturesToRow :: Matrix Double -> Integer -> Integer
  -> [Integer]

findClosestFeaturesToRow records row knn =
    take (fromIntegral knn) orderOfFeatures
  where
    sumOfSquares = toList $ euclideanDistanceSqrdFromRow records
    row
    orderOfFeatures = order sumOfSquares . nub $ sort
    sumOfSquares
```

Testing our recommendation engine

This aspect of the testing process will be different for everyone since each of us should have downloaded our databases individually. I found a user in my database at the 22951 index position who tweeted a lot. We are going to see whether we can recommend users to user #22951.

First, here's the feature vector for user #22951 representing the 25 most commonly used words in our database:

```
> genericIndex wordsMatrix 22951
[0.0,0.0,0.0,24.0,0.0,24.0,0.0,24.0,0.0,0.0,24.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
```

It seems that this user's word patterns focused on just 4 popular words. Let's use our recommendation to provide the 5 users who are the most similar to #22951 using the following statement. Note that we purposely typed `6` here instead of `5`. The closest match to #22951 is #22951, which shouldn't be surprising to us at all. (We will hope that a recommendation engine will be smart enough to notice someone identical to ourselves first.) Ignore the first result.

```
> findClosestFeaturesToRow pcaMatrix 22951 6
[22951,8020,13579,19982,11178,22227]
```

To see whether our engine works, the feature vector of each of these five users will hopefully be similar to #22951's feature vector. These users won't have the exact same word usage pattern as #22951, but it will be similar enough to provide a recommendation. This can be seen using the following statements:

```
> genericIndex wordsMatrix 8020
[0.0,0.0,0.0,0.0,0.0,14.0,0.0,14.0,0.0,0.0,0.0,14.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0]


> genericIndex wordsMatrix 13579
[0.0,0.0,0.0,0.0,0.0,16.0,0.0,8.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0]


> genericIndex wordsMatrix 19982
[0.0,0.0,13.0,0.0,0.0,13.0,0.0,13.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0]


> genericIndex wordsMatrix 11178
[10.0,0.0,0.0,0.0,0.0,10.0,0.0,10.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0
.0,0.0,0.0]


> genericIndex wordsMatrix 22227
[0.0,0.0,0.0,0.0,0.0,8.0,0.0,8.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0]
```

Our PCA recommendation engine was able to find some users that shared a common word usage pattern with the user at row 22951 in our database using only the eigenvectors associated with the top 5 eigenvalues. This was a small database of only 25,000 users with about 1 tweet each and we were still able to come up with a short list of similar users.

Summary

In this chapter, we used an analysis of the word frequency of tweets and the mathematics behind eigenvectors to build a recommendation engine to help Twitter users find other users similar to themselves. Along the way, we learned that there are many words in our language that aren't useful to the data analysis process. We filtered out the stop words from our dataset. We studied the jump from bivariate data to multivariate data and looked at the tools that linear algebra has to offer. Before this, we had worked with simple lists and a single input and output. We got the chance to work with matrices with many dimensions of data. We were able to organize our multidimensional datasets using a covariance matrix. We were able to measure the skewness of our covariance matrix using the eigenvalue decomposition process. We learned that not all dimensions of data are useful, and we were able to weed out the less useful dimensions by utilizing Principal Component Analysis (the process by which we used only the top ranked eigenvalues and eigenvectors). Using our lower dimensional dataset, we built a recommendation engine that searched for the users that were the nearest matches to a user using the Euclidean squared distance measure. Finally, we tested our dataset by selecting a user and discovered some similar users to this member.

Regular Expressions in Haskell

In *Chapter 3*, *Cleaning Our Datasets*, we discussed the use of regular expressions to assist the programmer in finding irregularities in datasets. *Chapter 3*, *Cleaning Our Datasets*, in my opinion, is the most important chapter in the book. This appendix sets out to compile a list of the most frequently used regular expression features. Before we begin, we must import the `regex-posix` package:

```
> import Text.Regex.Posix
```

A crash course in regular expressions

A regular expression is made up of atoms. An unmodified atom in a regular expression must match one, and exactly one, instance of a matching sequence in a string in order to satisfy the expression. When two or more unmodified atoms appear consecutively in an expression (such as `Jim` used in *Chapter 3*, *Cleaning Our Datasets*), the sequence of `J`, followed immediately by `i`, which is then followed immediately by `m`, must appear somewhere in the string. This behavior is similar to the **Find** feature found in most text editors and word processors. This is also where regular expressions begin to differ from a simple substring search. The sequence of `Jim` can be seen using the following statements:

```
> ("My name is Jim." =~ "Jim") :: Bool
True
> ("My name is Frank." =~ "Jim") :: Bool
False
```

The three repetition modifiers

Atoms can be modified. In our initial examples, we see our modifiers acting on a single character, but (as demonstrated later) an atom may be a single character or a grouping of characters. There are three primary modifiers in regular expressions. The `*` modifier means that an atom may be found zero or more times in a string. For example, `a*` can match `a`, `aaaaaa`, `rabbit`, and even `cow`. The `+` modifier means that an atom should be found one or more times. The `a+` expression can match `a`, `aaaaa`, and `rabbit`, but it will never match `cow`. Finally, there is `?`, which means that an atom may exist zero times or once (think of it as the `maybe` modifier). These atoms are known as greedy modifiers, which means that they try to gobble up as much of the string with an atom as possible. The `*` and `?` modifiers may both match with 0 instances of an atom. Since there is always going to be at least a zero match of everything, these modified atoms will always return a successful match unless joined by something else.

When using the `*` modifier, it is important to remember that the atom will always have a match and it will never be evaluated as `False`:

```
> ("a" =~ "a*") :: Bool
True
> ("aaaaaaaa" =~ "a*") :: Bool
True
> ("rabbit" =~ "a*") :: Bool
True
> ("cow" =~ "a*") :: Bool
True
> ("" =~ "a*") :: Bool
True
```

When using the `+` modifier, the atom must match at least one character in order to be `True`:

```
> ("a" =~ "a+") :: Bool
True
> ("aaaaaaaa" =~ "a+") :: Bool
True
> ("rabbit" =~ "a+") :: Bool
True
> ("cow" =~ "a+") :: Bool
False
> ("" =~ "a+") :: Bool
False
```

Depending on where you live in the world, the agreed-upon correct spelling of a term that is a synonym of hue can be `color` or `colour`. Using the `?` modifier, we can craft a regular expression that evaluates to `True` for both the spellings and to `False` for incorrect spellings. Like the `*` modifier, this atom modification will always evaluate to `True` when the expression matches the intended atom (and thus consumes the matched characters) or when it matches with nothing. The last expression in the following example evaluates to `False` due to other atoms in the expression:

```
> ("color" =~ "colou?r") :: Bool
True
> ("colour" =~ "colou?r") :: Bool
True
> ("coluor" =~ "colou?r") :: Bool
False
```

Anchors

Regular expressions can be anchored to either of the two ends of a string. The symbol for the start and end of the string is `^` and `$` respectively. Let's examine a regular expression to match with the words that contain `grand`, but only at the beginning of the expression:

```
> ("grandmother" =~ "^grand") :: Bool
True
> ("hundred grand" =~ "^grand") :: Bool
False
```

Likewise, we can anchor an expression to the end of a string. Let's examine the expression to match words ending with the `-ing` suffix:

```
> ("writing" =~ "ing$") :: Bool
True
> ("zingers" =~ "ing$") :: Bool
False
```

We can use `^` and `$` together if you want to match a regular expression with a word that may have the expression stretching across the entire string. When using the anchors together, the regular expression engine expects the regular expression to match from the start of the string to the end of the string. The first occurrence of an atom that evaluates to `False` will cause the entire expression to be evaluated to `False`. The last example in the following set evaluates to `True` because the string starts and ends with 0 occurrences of `a`:

```
> ("a" =~ "^a*$") :: Bool
True
> ("aaaaaaaa" =~ "^a*$") :: Bool
True
> ("rabbit" =~ "^a*$") :: Bool
False
> ("cow" =~ "^a*$") :: Bool
False
> ("" =~ "^a*$") :: Bool
True
```

The dot

The dot (or the period) is a special atom that allows us to match any one character. With the modification of `*` to make `.*`, we have crafted an expression that will match everything. To craft an expression that matches only a period, you have to escape the dot with a `\` for the regular expression engine. Since Haskell escapes strings before passing them to the regular expression engine, we have to escape the `\` with a second `\`:

```
> ("." =~ ".") :: Bool
True
> ("a" =~ ".") :: Bool
True
> ("." =~ "\\.") :: Bool
True
> ("a" =~ "\\.") :: Bool
False
```

Character classes

Characters can be grouped into character classes using the square brackets, `[` and `]`. A character class is a collection of characters in which one character must match. The words `grey` and `gray` can be considered correct spellings. We can craft a regular expression to match both spellings:

```
> ("grey" =~ "gr[ae]y") :: Bool
True
> ("gray" =~ "gr[ae]y") :: Bool
True
> ("graey" =~ "gr[ae]y") :: Bool
False
```

By beginning a character class with `^`, we create the complement of a character class. That is, a character will match because it is not found in that character class. For example, we can check to see whether a word doesn't contain any vowels using a regular expression. This requires us to modify the character class so that it matches at least one character using `+` and is anchored to the beginning and end of the expression:

```
> ("rabbit" =~ "^[^aeiou]+$") :: Bool
False
> ("cow" =~ "^[^aeiou]+$") :: Bool
False
> ("why" =~ "^[^aeiou]+$") :: Bool
True
```

Character classes can also support a range of letters. Rather than requiring a character class to match a lowercase letter that looks like `[abcdefghijklmnopqrstuvwxyz]`, it is clearer to write `[a-z]`. The `[A-Z]` range works for uppercase letters and `[0-9]` works for numbers:

```
> ("a" =~ "[a-z]") :: Bool
True
> ("A" =~ "[a-z]") :: Bool
False
> ("s" =~ "[A-Z]") :: Bool
False
> ("S" =~ "[A-Z]") :: Bool
True
> ("S" =~ "[a-zA-Z]") :: Bool
True
```

Groups

Atoms can be grouped together using parentheses, and these groups can be modified. Therefore, the regular expression `(row, )+row your boat` will match the entire string of `row, row, row your boat`. An added benefit of grouping is that the text matched in one part of an expression can be used as a regular expression that is used later on in the same expression:

```
> ("row your boat" =~ "(row, )+row your boat") :: Bool
False
> ("row, row your boat" =~ "(row, )+row your boat") :: Bool
True
> ("row, row, row your boat" =~ "(row, )+row your boat") :: Bool
True
> ("row, row, row, row your boat" =~ "(row, )+row your boat") :: Bool
True
```

The lyrics to this song are `row, row, row your boat`. We can enforce that there are exactly three `row` words in our string (two with commas after it and one without). We also need to use our anchors and the `{}` modifier, which enforces an explicit number of repetitions:

```
> ("row your boat" =~ "^(row, ){2}row your boat$") :: Bool
False
> ("row, row your boat" =~ "^(row, ){2}row your boat$") :: Bool
False
> ("row, row, row your boat" =~ "^(row, ){2}row your boat$") :: Bool
True
> ("row, row, row, row your boat" =~ "^(row, ){2}row your boat$") :: Bool
False
```

Alternations

Alterations can happen any time we want one expression or another. For example, we wish to create a regular expression to match the year of birth of someone that is still alive. At the time of writing this book, the oldest living person was born in 1899. We would like to craft a regular expression to match the birth year of anyone born after 1899 to 2099. We can do this with an alternation.

Using the pipe character `|`, we can say that the regular expression `A|B|C` must match `A`, `B`, or `C` in order to be evaluated as `True`. Now, we must craft three separate regular expressions for the year `1899`, any year in the 1900s, and any year in the 2000s:

```
> ("1898" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
False
> ("1899" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
True
> ("1900" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
True
> ("1999" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
True
> ("2015" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
True
> ("2115" =~ "^1899|19[0-9][0-9]|20[0-9][0-9]$") :: Bool
False
```

A note on regular expressions

Regular expressions are defined by their engine, and every regular expression engine has differences. We did our best in this appendix to include features of regular expressions that are common across most engines. (When creating this appendix, we discovered differences between the `regex-posix` package on Windows and Linux.) Some excellent resources to learn about regular expressions include *Mastering Regular Expressions*, *Jeffrey Friedl*, *O'Reilly Media* and *Mastering Python Regular Expressions*, *Felix Lopez*, *Victor Romero*, *Packt Publishing*.

Regular expressions should be avoided whenever possible. They are difficult to read, debug, and test, and are prone to being slow. Sometimes, a parser is a better solution than a regular expression due to the recursive nature of some text. If you find a regular expression on the Internet that you intend to use in your code, be sure to test it thoroughly. If you can find a function that does something similar to your needs without having to craft a regular expression, I recommend that you use the function instead.

Having said why you shouldn't use regular expressions, I believe that they provide a fun and intellectual challenge to craft expressions to match patterns of text. A simple regular expression will help you find features in your datasets, which is easier than a simple substring search.

Index

M

N

O

P

R

S

T

U

V

W

Y

Thank you for buying
Learning Haskell Data Analysis

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

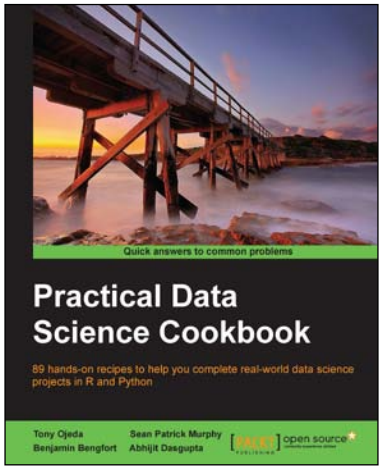
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

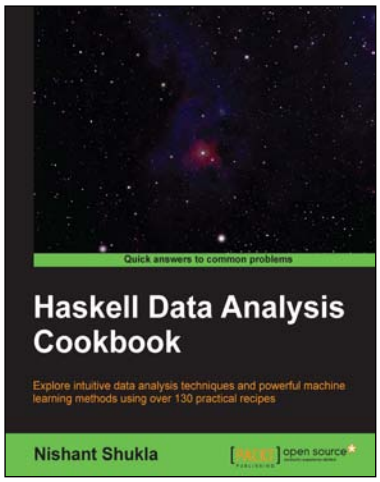


Practical Data Science Cookbook

ISBN: 978-1-78398-024-6 Paperback: 396 pages

89 hands-on recipes to help you complete real-world data science projects in R and Python

1. Learn about the data science pipeline and use it to acquire, clean, analyze, and visualize data.
2. Understand critical concepts in data science in the context of multiple projects.
3. Expand your numerical programming skills through step-by-step code examples and learn more about the robust features of R and Python.

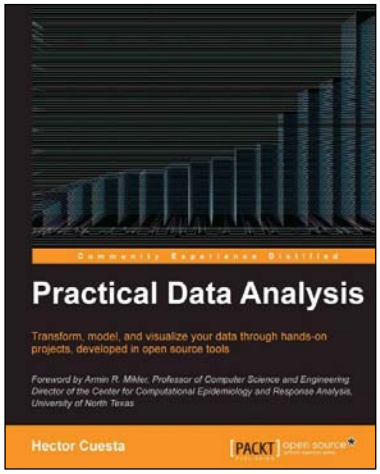


Haskell Data Analysis Cookbook

ISBN: 978-1-78328-633-1 Paperback: 334 pages

Explore intuitive data analysis techniques and powerful machine learning methods using over 130 practical recipes

1. A practical and concise guide to using Haskell when getting to grips with data analysis.
2. Recipes for every stage of data analysis, from collection to visualization.
3. In-depth examples demonstrating various tools, solutions and techniques.

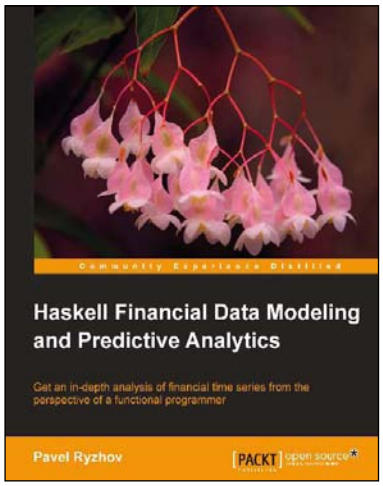


Practical Data Analysis

ISBN: 978-1-78328-099-5 Paperback: 360 pages

Transform, model, and visualize your data through hands-on projects, developed in open source tools

1. Explore how to analyze your data in various innovative ways and turn them into insight.
2. Learn to use the D3.js visualization tool for exploratory data analysis.
3. Understand how to work with graphs and social data analysis.
4. Discover how to perform advanced query techniques and run MapReduce on MongoDB.



Haskell Financial Data Modeling and Predictive Analytics

ISBN: 97978-1-78216-943-7 Paperback: 112 pages

Get an in-depth analysis of financial time series from the perspective of a functional programmer

1. Understand the foundations of financial stochastic processes.
2. Build robust models quickly and efficiently.
3. Tackle the complexity of parallel programming.