Daniel Whitenack

Machine Learning With Go

Implement Regression, Classification, Clustering, Time-series Models, Neural Networks, and More using the Go Programming Language



Packt>

Machine Learning With Go

Implement Regression, Classification, Clustering, Time-series Models, Neural Networks, and More using the Go Programming Language

Daniel Whitenack



BIRMINGHAM - MUMBAI

Machine Learning With Go

Copyright © 2017 Packt Publishing

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

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

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

First published: September 2017

Production reference: 1210917

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

ISBN 978-1-78588-210-4

www.packtpub.com

Credits

Author
Daniel Whitenack

Reviewers
Niclas Jern
Richard Townsend

Commissioning Editor
Veena Pagare

Acquisition Editor
Varsha Shetty

Content Development Editor
Snehal Kolte

Technical Editor
Sagar Sawant

Copy Editor
Tasneem Fatehi

Project Coordinator
Manthan Patel

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Tania Dutta

Production Coordinator
Deepika Naik

About the Author

Daniel Whitenack (`@dwhitena`), PhD, is a trained data scientist working with Pachyderm (`@pachydermIO`). Daniel develops innovative, distributed data pipelines that include predictive models, data visualizations, statistical analyses, and more. He has spoken at conferences around the world (GopherCon, JuliaCon, PyCon, ODSC, Spark Summit, and more), teaches data science/engineering at Purdue University (`@LifeAtPurdue`), and, with Ardan Labs (`@ardanlabs`), maintains the Go kernel for Jupyter, and is actively helping to organize contributions to various open source data science projects.

I would like to thank my wife for her boundless patience and support while writing this book.
I would also like to acknowledge the many wonderful gophers and data scientists that have mentored me, collaborated with me, and encouraged me. These include Bill Kennedy, Brendan Tracey, Sebastien Binet, Alex Sanchez, the whole team at Pachyderm (including Joey Zwicker and Joe Doliner), Chris Tava, Mat Ryer, David Hernandez, Xuanyi Chew, the team at Minio (including Anand Babu Periasamy and Garima Kapoor), and many more!

Soli Deo Gloria

About the Reviewers

Niclas Jern has been using Go to solve interesting problems at scale since Go 1.0. He graduated from Abo Akademi University with an MSc in computer engineering, majoring in software engineering.

He enjoys using Go and other programming languages to tackle problems, especially in the fields of data processing and machine learning. His hobbies include long walks, lifting heavy metal objects at the gym, the occasional rant at `http://www.njern.co`, and spending quality time with his wife and daughter.

Niclas currently works at Walkbase, a company he founded together with some of his class mates from Abo Akademi university, where he leads the engineering team, which is tackling the data processing problems that come with revolutionizing retail analytics.

Richard Townsend became the top contributor to GoLearn in 2014 (and hence is responsible for a lot of its odd behavior) while studying for his undergraduate degree at Warwick University. Since then, he's worked for a top UK technology company on everything from embedded systems to Android operating system frameworks, and currently spends his time optimizing web browsers. He still spends a significant amount of time on sentiment analysis (co-authoring two papers on it) and other natural language processing tasks like part of speech tagging often using the latest deep learning technologies.

www.PacktPub.com

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



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

Why subscribe?

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1785882104`.

If you'd like to join our team of regular reviewers, you can email us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface 1

Chapter 1: Gathering and Organizing Data 8

- **Handling data - Gopher style** 9
- **Best practices for gathering and organizing data with Go** 11
- **CSV files** 12
  - Reading in CSV data from a file 13
  - Handling unexpected fields 14
  - Handling unexpected types 15
  - Manipulating CSV data with data frames 18
- **JSON** 20
  - Parsing JSON 20
  - JSON output 23
- **SQL-like databases** 24
  - Connecting to an SQL database 24
  - Querying the database 25
  - Modifying the database 26
- **Caching** 27
  - Caching data in memory 28
  - Caching data locally on disk 28
- **Data versioning** 30
  - Pachyderm jargon 31
  - Deploying/installing Pachyderm 31
  - Creating data repositories for data versioning 32
  - Putting data into data repositories 33
  - Getting data out of versioned data repositories 33
- **References** 33
- **Summary** 34

Chapter 2: Matrices, Probability, and Statistics 35

- **Matrices and vectors** 35
  - Vectors 36
  - Vector operations 36
  - Matrices 38
  - Matrix operations 40
- **Statistics** 41

Distributions 42
Statistical measures 43
Measures of central tendency 43
Measures of spread or dispersion 45
Visualizing distributions 48
Histograms 48
Box plots 52
Probability 55
Random variables 55
Probability measures 56
Independent and conditional probability 56
Hypothesis testing 57
Test statistics 58
Calculating p-values 58
References 60
Summary 61
Chapter 3: Evaluation and Validation 62
Evaluation 62
Continuous metrics 63
Categorical metrics 67
Individual evaluation metrics for categorical variables 67
Confusion matrices, AUC, and ROC 72
Validation 75
Training and test sets 76
Holdout set 80
Cross validation 81
References 83
Summary 84
Chapter 4: Regression 85
Understanding regression model jargon 85
Linear regression 86
Overview of linear regression 87
Linear regression assumptions and pitfalls 89
Linear regression example 90
Profiling the data 90
Choosing our independent variable 94
Creating our training and test sets 97
Training our model 99
Evaluating the trained model 100
Multiple linear regression 104
Nonlinear and other types of regression 109

References 114
Summary 114
Chapter 5: Classification 115
Understanding classification model jargon 116
Logistic regression 116
Overview of logistic regression 117
Logistic regression assumptions and pitfalls 121
Logistic regression example 121
Cleaning and profiling the data 122
Creating our training and test sets 127
Training and testing the logistic regression model 129
k-nearest neighbors 135
Overview of kNN 135
kNN assumptions and pitfalls 137
kNN example 138
Decision trees and random forests 140
Overview of decision trees and random forests 140
Decision tree and random forest assumptions and pitfalls 141
Decision tree example 142
Random forest example 143
Naive bayes 144
Overview of naive bayes and its big assumption 144
Naive bayes example 144
References 146
Summary 147
Chapter 6: Clustering 148
Understanding clustering model jargon 149
Measuring Distance or Similarity 149
Evaluating clustering techniques 151
Internal clustering evaluation 151
External clustering evaluation 156
k-means clustering 157
Overview of k-means clustering 157
k-means assumptions and pitfalls 160
k-means clustering example 161
Profiling the data 161
Generating clusters with k-means 164
Evaluating the generated clusters 166
Other clustering techniques 169

References 170
Summary 170
Chapter 7: Time Series and Anomaly Detection 171
Representing time series data in Go 172
Understanding time series jargon 175
Statistics related to time series 176
Autocorrelation 176
Partial autocorrelation 181
Auto-regressive models for forecasting 184
Auto-regressive model overview 184
Auto-regressive model assumptions and pitfalls 185
Auto-regressive model example 186
Transforming to a stationary series 186
Analyzing the ACF and choosing an AR order 189
Fitting and evaluating an AR(2) model 190
Auto-regressive moving averages and other time series models 196
Anomaly detection 197
References 199
Summary 199
Chapter 8: Neural Networks and Deep Learning 201
Understanding neural net jargon 202
Building a simple neural network 203
Nodes in the network 204
Network architecture 206
Why do we expect this architecture to work? 207
Training our neural network 208
Utilizing the simple neural network 215
Training the neural network on real data 216
Evaluating the neural network 218
Introducing deep learning 220
What is a deep learning model? 221
Deep learning with Go 222
Setting up TensorFlow for use with Go 224
Retrieving and calling a pretrained TensorFlow model 224
Object detection using TensorFlow from Go 226
References 230
Summary 230
Chapter 9: Deploying and Distributing Analyses and Models 231
Running models reliably on remote machines 232

A brief introduction to Docker and Docker jargon 232
Docker-izing a machine learning application 234
Docker-izing the model training and export 234
Docker-izing model predictions 239
Testing the Docker images locally 244
Running the Docker images on remote machines 246
Building a scalable and reproducible machine learning pipeline 247
Setting up a Pachyderm and Kubernetes cluster 248
Building a Pachyderm machine learning pipeline 250
Creating and filling the input repositories 251
Creating and running the processing stages 255
Updating pipelines and examining provenance 260
Scaling pipeline stages 262
References 264
Summary 265
Chapter 10: Algorithms/Techniques Related to Machine Learning 266
Gradient descent 266
Entropy, information gain, and related methods 269
Backpropagation 271
Index 275

Preface

It seems like machine learning and artificial intelligence is all the rage, both in hip tech companies and increasingly in larger enterprise companies. Data scientists are using machine learning to do everything from drive cars to draw cats. However, if you follow the data science community, you have very likely seen something like *language wars* unfold between Python and R users. These languages dominate the machine learning conversation and often seem to be the only choices to integrate machine learning in your organization. We will explore a third option in this book: Go, the open source programming language created at Google.

The unique features of Go, along with the mindset of Go programmers, can help data scientists overcome some of the common struggles that they encounter. In particular, data scientists are (unfortunately) known to produce bad, inefficient, and unmaintainable code. This book will address this issue, and will clearly show you how to be productive in machine learning while also producing applications that maintain a high level of integrity. It will also allow you to overcome the common challenges of integrating analysis and machine learning code within an existing engineering organization.

This book will develop readers into productive, innovative data analysts who leverage Go to build robust and valuable applications. To this end, the book will clearly introduce the technical, programming aspects of machine learning in Go, but it will also guide the reader to understand sound workflows and philosophies for real-world analysis.

What this book covers

Preparing and analyzing data in machine learning workflows:

- `Chapter 1`, *Gathering and Organizing Data*, covers the gathering, organization, and parsing of data to/from a local and remote sources. Once the reader is done with this chapter, they will understand how to interact with data stored in various places and in various formats, how to parse and clean that data, and how to output that cleaned and parsed data.

- `Chapter 2`, *Matrices, Probability, and Statistics*, covers the organization of data into matrices and matrix operations. Once the reader is done with this material, they will understand how to form matrices within Go programs and how to utilize these matrices to perform various types of matrix operations. This chapter also covers statistical measures and operations key to day-to-day data analysis work. Once the reader is done with this chapter, they will understand how to perform solid summary data analysis, describe and visualize distributions, quantify hypotheses, and transform datasets with, for example, dimensionality reductions.
- `Chapter 3`, *Evaluation and Validation*, covers evaluation and validation, which are key to measuring the performance of machine applications and ensuring that they generalize. Once the reader is done with this chapter, they will understand various metrics to gauge the performance of models (in other words, **evaluate** the model) as well as various techniques to validate the model more generally.

Machine learning techniques:

- `Chapter 4`, *Regression*, explains regression, a widely used technique to model continuous variables and a basis for other models. Regression produces models that are immediately interpretable. Thus, it can provide an excellent starting point when introducing predictive capabilities in a organization.
- `Chapter 5`, *Classification*, covers classification, a machine learning technique distinct from regression in that the target variable is typically categorical or labeled. For example, a classification model may classify emails into spam and not spam categories or classify network traffic as fraudulent or not fraudulent.
- `Chapter 6`, *Clustering*, covers clustering, an unsupervised machine learning technique used to form groupings of samples. At the end of this chapter, readers will be able to automatically form groupings of data points to better understand their structure.
- `Chapter 7`, *Time Series and Anomaly Detection*, introduces techniques utilized to model time series data, such as stock prices, user events, and so on. After reading this chapter, the reader will understand how to evaluate various terms in a time series, build up a model of the time series, and detect anomalies in a time series.

Taking machine learning to the next level:

- `Chapter 8`, *Neural Networks and Deep Learning*, introduces techniques utilized to perform regression, classification, and image processing with neural networks. After reading this chapter, the reader will understand how and when to apply these more complicated modeling techniques.

- `Chapter 9`, *Deploying and Distributing Analyses and Models*, empowers readers to deploy the models that we have developed throughout the class to production environments and distribute processing over production scale data. This chapter illustrates how both of these things can be done easily, without significant modifications to the code utilized throughout the book.

The `Appendix`, *Algorithms/Techniques Related to Machine Learning*, can be referenced throughout the text of the book and will provide information about algorithms, optimizations, and techniques that are relevant to machine learning workflows.

What you need for this book

To run the examples in this book and experiment with the techniques covered in the book, you will generally need the following:

- Access to a bash-like shell.
- A complete Go environment including Go, an editor, and related default or custom environment variables defined. You can, for example, follow this guide at `https://www.goinggo.net/2016/05/installing-go-and-your-workspace.html`.
- Various Go dependencies. These can be obtained as they are needed via `go get ...`.

Then, to run the examples related to some of the advanced topics, such as data pipelining and deep learning, you will need a few additional things:

- An installation or deployment of Pachyderm. You can follow these docs to get Pachyderm up and running locally or in the cloud, `http://pachyderm.readthedocs.io/en/latest/`.
- A working Docker installation (`https://www.docker.com/community-edition#/download`).
- An installation of TensorFlow. To install TensorFlow locally, you can follow this guide at `https://www.tensorflow.org/install/`.

Who this book is for

If you are one of the following, this book is going to be useful for you:

1. A Go programmer who is interested in machine learning and data analysis
2. A data scientist / analyst / engineer who has interest in Go and wants to integrate it in their machine learning and data analysis workflow

Conventions

Many Go code snippets will not include `package main` and `func main() {}`. It is assumed that Go code snippets are compiled within this necessary structure, unless otherwise noted. It will also be assumed throughout the book that code example are compiled in a directory call `myprogram` and thus compiled to a `myprogram` binary. The reader will recognize, however, that the code could be copied to any folder in your `GOPATH/src` directory and/or compiled to a binary named according to the reader's preference.

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the to the `BeautifulSoup` function." A block of code is set as follows:

```
#import packages into the project
from bs4 import BeautifulSoup
from urllib.request import urlopen
import pandas as pd
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
 [default] exten => s,1,Dial(Zap/1|30)
 exten => s,2,Voicemail(u100)
 exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
C:\Python34\Scripts> pip install –upgrade pip
C:\Python34\Scripts> pip install pandas
```

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



Warnings or important notes appear 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 email `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 original code files from the date of publication by visiting your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

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

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Machine-Learning-With-Go`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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/MachineLearningWithGo_ColorImages.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
Gathering and Organizing Data

Polls have shown that 90% or more of a data scientist's time is spent gathering data, organizing it, and cleaning it, not training/tuning their sophisticated machine learning models. Why is this? Isn't the machine learning part the fun part? Why do we need to care so much about the state of our data? Firstly, without data, our machine learning models can't learn. This might seem obvious. However, we need to realize that part of the strength of the models that we build is in the data that we feed them. As the common phrase goes, *garbage in, garbage out*. We need to make sure that we gather relevant, clean data to power our machine learning models, such that they can operate on the data as expected and produce valuable results.

Not all types of data are appropriate when using certain types of models. For example, certain models do not perform well when we have high-dimensional data (for example, text data), and other models assume that variables are normally distributed, which is definitely not always the case. Thus, we must take care in gathering data that fits our use case and make sure that we understand how our data and models will interact.

Another reason why gathering and organizing data consumes so much of a data scientist's time is that data is often messy and hard to aggregate. In most organizations, data might be housed in various systems and formats, and have various access control policies. We can't assume that supplying a training set to our model will be as easy as specifying a file path; this is often not the case.

To form a training/test set or to supply variables to a model for predictions, we will likely need to deal with various formats of data, such as CSV, JSON, database tables, and so on, and we will likely need to transform individual values. Common transformations include parsing date times, converting categorical data to numerical data, normalizing values, and applying some function across values. However, we can't always assume that all values of a certain variable are present or able to be parsed in a similar manner.

Often data includes missing values, mixed types, or corrupted values. How we handle each of these scenarios will directly influence the quality of the models that we build, and thus, we have to be willing to carefully gather, organize, and understand our data.

Even though much of this book will be focused on various modeling techniques, you should always consider data gathering, parsing, and organization as a (or maybe the) key component of a successful data science project. If this part of your project is not carefully developed with a high level of integrity, you are setting yourself up for trouble in the long run.

Handling data - Gopher style

In comparison to many other languages that are used for data science/analysis, Go provides a very strong foundation for data manipulation and parsing. Although other languages (for example, Python or R) may allow users to quickly explore data interactively, they often promote integrity-breaking convenience, that is, dynamic and interactive data exploration often results in code that behaves strangely when applied more generally.

Take, for instance, this simple CSV file:

```
1,blah1
2,blah2
3,blah3
```

It is true that, very quickly, we can write some Python code to parse this CSV and output the maximum value from the integer column without even knowing what types are in the data:

```
import pandas as pd

# Define column names.
cols = [
 'integercolumn',
 'stringcolumn'
 ]

# Read in the CSV with pandas.
data = pd.read_csv('myfile.csv', names=cols)

# Print out the maximum value in the integer column.
print(data['integercolumn'].max())
```

This simple program will print the correct result:

```
$ python myprogram.py
3
```

We now remove one of the integer values to produce a missing value, as shown here:

```
1,blah1
2,blah2
,blah3
```

The Python program consequently has a complete breakdown in integrity; specifically, the program still runs, doesn't tell us that anything went differently, still produces a value, and produces a value of a different type:

```
$ python myprogram.py
2.0
```

This is unacceptable. All but one of our integer values could disappear, and we wouldn't have any insight into the changes. This could produce profound changes in our modeling, but they would be extremely hard to track down. Generally, when we opt for the conveniences of dynamic types and abstraction, we are accepting this sort of variability in behavior.



The important thing here is not that you cannot handle such behavior in Python, because Python, experts will quickly recognize that you can properly handle such behavior. The point is that such conveniences do not promote integrity by default, and thus, it is very easy to shoot yourself in the foot.

On the other hand, we can leverage Go's static typing and explicit error handling to ensure that our data is parsed as expected. In this small example, we can also write some Go code, without too much trouble, to parse our CSV (don't worry about the details right now):

```
// Open the CSV.
f, err := os.Open("myfile.csv")
if err != nil {
    log.Fatal(err)
}

// Read in the CSV records.
r := csv.NewReader(f)
records, err := r.ReadAll()
if err != nil {
    log.Fatal(err)
}
```

```
// Get the maximum value in the integer column.
var intMax int
for _, record := range records {

    // Parse the integer value.
    intVal, err := strconv.Atoi(record[0])
    if err != nil {
        log.Fatal(err)
    }

    // Replace the maximum value if appropriate.
    if intVal > intMax {
        intMax = intVal
    }
}

// Print the maximum value.
fmt.Println(intMax)
```

This will produce the same correct result for the CSV file with all the integer values present:

```
$ go build
$ ./myprogram
3
```

But in contrast to our previous Python code, our Go code will inform us when we encounter something that we don't expect in the input CSV (for the case when we remove the value 3):

```
$ go build
$ ./myprogram
2017/04/29 12:29:45 strconv.ParseInt: parsing "": invalid syntax
```

Here, we have maintained integrity, and we can ensure that we can handle missing values in a manner that is appropriate for our use case.

Best practices for gathering and organizing data with Go

As you can see in the preceding section, Go itself provides us with an opportunity to maintain high levels of integrity in our data gathering, parsing, and organization. We want to ensure that we leverage Go's unique properties whenever we are preparing our data for machine learning workflows.

Generally, Go data scientists/analysts should follow the following best practices when gathering and organizing data. These best practices are meant to help you maintain integrity in your applications, and been able you to reproduce any analysis:

1. **Check for and enforce expected types**: This might seem obvious, but it is too often overlooked when using dynamically typed languages. Although it is slightly verbose, explicitly parsing data into expected types and handling related errors can save you big headaches down the road.
2. **Standardize and simplify your data ingress/egress**: There are many third-party packages for handling certain types of data or interactions with certain sources of data (some of which we will cover in this book). However, if you standardize the ways you are interacting with data sources, particularly centered around the use of `stdlib`, you can develop predictable patterns and maintain consistency within your team. A good example of this is a choice to utilize `database/sql` for database interactions rather than using various third-party APIs and DSLs.
3. **Version your data**: Machine learning models produce extremely different results depending on the training data you use, your choice of parameters, and input data. Thus, it is impossible to reproduce results without versioning both your code and data. We will discuss the appropriate techniques for data versioning later in this chapter.



If you start to stray from these general principles, you should stop immediately. You are likely to sacrifice integrity for the sake of convenience, which is a dangerous road. We will let these principles guide us through the book and as we consider various data formats/sources in the following section.

CSV files

CSV files might not be a go-to format for big data, but as a data scientist or developer working in machine learning, you are sure to encounter this format. You might need a mapping of zip codes to latitude/longitude and find this as a CSV file on the internet, or you may be given sales figures from your sales team in a CSV format. In any event, we need to understand how to parse these files.

The main package that we will utilize in parsing CSV files is `encoding/csv` from Go's standard library. However, we will also discuss a couple of packages that allow us to quickly manipulate or transform CSV data--`github.com/kniren/gota/dataframe` and `go-hep.org/x/hep/csvutil`.

Reading in CSV data from a file

Let's consider a simple CSV file, which we will return to later, named `iris.csv` (available here: `https://archive.ics.uci.edu/ml/datasets/iris`). This CSV file includes four float columns of flower measurements and a string column with the corresponding flower species:

```
$ head iris.csv
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

With `encoding/csv` imported, we first open the CSV file and create a CSV reader value:

```
// Open the iris dataset file.
f, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
```

Then we can read in all of the records (corresponding to rows) of the CSV file. These records are imported as `[][]string`:

```
// Assume we don't know the number of fields per line. By setting
// FieldsPerRecord negative, each row may have a variable
// number of fields.
reader.FieldsPerRecord = -1

// Read in all of the CSV records.
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}
```

We can also read in records one at a time in an infinite loop. Just make sure that you check for the end of the file (`io.EOF`) so that the loop ends after reading in all of your data:

```
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = -1

// rawCSVData will hold our successfully parsed rows.
var rawCSVData [][]string

// Read in the records one by one.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Append the record to our dataset.
    rawCSVData = append(rawCSVData, record)
}
```



If your CSV file is not delimited by commas and/or if your CSV file contains commented rows, you can utilize the `csv.Reader.Comma` and `csv.Reader.Comment` fields to properly handle uniquely formatted CSV files. In cases where the fields in your CSV file are single-quoted, you may need to add in a helper function to trim the single quotes and parse the values.

Handling unexpected fields

The preceding methods work fine with clean CSV data, but, in general, we don't encounter clean data. We have to parse messy data. For example, you might find unexpected fields or numbers of fields in your CSV records. This is why `reader.FieldsPerRecord` exists. This field of the reader value lets us easily handle messy data, as follows:

```
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,blah,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
```

This version of the `iris.csv` file has an extra field in one of the rows. We know that each record should have five fields, so let's set our `reader.FieldsPerRecord` value to `5`:

```
// We should have 5 fields per line. By setting
// FieldsPerRecord to 5, we can validate that each of the
// rows in our CSV has the correct number of fields.
reader.FieldsPerRecord = 5
```

Then as we are reading in records from the CSV file, we can check for unexpected fields and maintain the integrity of our data:

```
// rawCSVData will hold our successfully parsed rows.
var rawCSVData [][]string

// Read in the records looking for unexpected numbers of fields.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // If we had a parsing error, log the error and move on.
    if err != nil {
        log.Println(err)
        continue
    }

    // Append the record to our dataset, if it has the expected
    // number of fields.
    rawCSVData = append(rawCSVData, record)
}
```

Here, we have chosen to handle the error by logging the error, and we only collect successfully parsed records into `rawCSVData`. The reader will note that this error could be handled in many different ways. The important thing is that we are forcing ourselves to check for an expected property of the data and increasing the integrity of our application.

Handling unexpected types

We just saw that CSV data is read into Go as `[][]string`. However, Go is statically typed, which allows us to enforce strict checks for each of the CSV fields. We can do this as we parse each field for further processing. Consider some messy data that has random fields that don't match the type of the other values in a column:

```
4.6,3.1,1.5,0.2,Iris-setosa
5.0,string,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
4.9,3.1,1.5,0.1,Iris-setosa
5.0,3.2,1.2,string,Iris-setosa
5.5,3.5,1.3,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
4.4,3.0,1.3,0.2,Iris-setosa
```

To check the types of the fields in our CSV records, let's create a `struct` variable to hold successfully parsed values:

```
// CSVRecord contains a successfully parsed row of the CSV file.
type CSVRecord struct {
    SepalLength  float64
    SepalWidth   float64
    PetalLength  float64
    PetalWidth   float64
    Species      string
    ParseError   error
}
```

Then, before we loop over the records, let's initialize a slice of these values:

```
// Create a slice value that will hold all of the successfully parsed
// records from the CSV.
var csvData []CSVRecord
```

Now as we loop over the records, we can parse into the relevant type for that record, catch any errors, and log as needed:

```
// Read in the records looking for unexpected types.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Create a CSVRecord value for the row.
    var csvRecord CSVRecord
```

```
        // Parse each of the values in the record based on an expected type.
        for idx, value := range record {

            // Parse the value in the record as a string for the string column.
            if idx == 4 {

                // Validate that the value is not an empty string. If the
                // value is an empty string break the parsing loop.
                if value == "" {
                    log.Printf("Unexpected type in column %d\n", idx)
                    csvRecord.ParseError = fmt.Errorf("Empty string value")
                    break
                }

                // Add the string value to the CSVRecord.
                csvRecord.Species = value
                continue
            }

            // Otherwise, parse the value in the record as a float64.
            var floatValue float64

            // If the value can not be parsed as a float, log and break the
            // parsing loop.
            if floatValue, err = strconv.ParseFloat(value, 64); err != nil {
                log.Printf("Unexpected type in column %d\n", idx)
                csvRecord.ParseError = fmt.Errorf("Could not parse float")
                break
            }

            // Add the float value to the respective field in the CSVRecord.
            switch idx {
            case 0:
                csvRecord.SepalLength = floatValue
            case 1:
                csvRecord.SepalWidth = floatValue
            case 2:
                csvRecord.PetalLength = floatValue
            case 3:
                csvRecord.PetalWidth = floatValue
            }
        }

        // Append successfully parsed records to the slice defined above.
        if csvRecord.ParseError == nil {
            csvData = append(csvData, csvRecord)
        }
    }
```

Manipulating CSV data with data frames

As you can see, manually parsing many different fields and performing row-by-row operations can be rather verbose and tedious. This is definitely *not* an excuse to increase complexity and import a bunch of non standard functionalities. You should still default to the use of `encoding/csv` in most cases.

However, manipulation of data frames has proven to be a successful and somewhat standardized way (in the data science community) of dealing with tabular data. Thus, in some cases, it is worth employing some third-party functionality to manipulate tabular data, such as CSV data. For example, data frames and the corresponding functionality can be very useful when you are trying to filter, subset, and select portions of tabular datasets. In this section, we will introduce `github.com/kniren/gota/dataframe`, a wonderful `dataframe` package for Go:

```
import "github.com/kniren/gota/dataframe"
```

To create a data frame from a CSV file, we open a file with `os.Open()` and then supply the returned pointer to the `dataframe.ReadCSV()` function:

```
// Open the CSV file.
irisFile, err := os.Open("iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
irisDF := dataframe.ReadCSV(irisFile)

// As a sanity check, display the records to stdout.
// Gota will format the dataframe for pretty printing.
fmt.Println(irisDF)
```

If we compile and run this Go program, we will see a nice, pretty-printed version of our data with the types that were inferred during parsing:

```
$ go build
$ ./myprogram
[150x5] DataFrame

 sepal_length sepal_width petal_length petal_width species
 0: 5.100000 3.500000 1.400000 0.200000 Iris-setosa
 1: 4.900000 3.000000 1.400000 0.200000 Iris-setosa
 2: 4.700000 3.200000 1.300000 0.200000 Iris-setosa
```

```
3: 4.600000 3.100000 1.500000 0.200000 Iris-setosa
4: 5.000000 3.600000 1.400000 0.200000 Iris-setosa
5: 5.400000 3.900000 1.700000 0.400000 Iris-setosa
6: 4.600000 3.400000 1.400000 0.300000 Iris-setosa
7: 5.000000 3.400000 1.500000 0.200000 Iris-setosa
8: 4.400000 2.900000 1.400000 0.200000 Iris-setosa
9: 4.900000 3.100000 1.500000 0.100000 Iris-setosa
... ... ... ... ...
<float> <float> <float> <float> <string>
```

Once we have the data parsed into a `dataframe`, we can filter, subset, and select our data easily:

```
// Create a filter for the dataframe.
filter := dataframe.F{
    Colname: "species",
    Comparator: "==",
    Comparando: "Iris-versicolor",
}

// Filter the dataframe to see only the rows where
// the iris species is "Iris-versicolor".
versicolorDF := irisDF.Filter(filter)
if versicolorDF.Err != nil {
    log.Fatal(versicolorDF.Err)
}

// Filter the dataframe again, but only select out the
// sepal_width and species columns.
versicolorDF = irisDF.Filter(filter).Select([]string{"sepal_width",
"species"})

// Filter and select the dataframe again, but only display
// the first three results.
versicolorDF = irisDF.Filter(filter).Select([]string{"sepal_width",
"species"}).Subset([]int{0, 1, 2})
```

This is really only scratching the surface of the `github.com/kniren/gota/dataframe` package. You can merge datasets, output to other formats, and even process JSON data. For more information about this package, you should visit the auto generated GoDocs at `https://godoc.org/github.com/kniren/gota/dataframe`, which is good practice, in general, for any packages we discuss in the book.

JSON

In a world in which the majority of data is accessed via the web, and most engineering organizations implement some number of microservices, we are going to encounter data in JSON format fairly frequently. We may only need to deal with it when pulling some random data from an API, or it might actually be the primary data format that drives our analytics and machine learning workflows.

Typically, JSON is used when ease of use is the primary goal of data interchange. Since JSON is human readable, it is easy to debug if something breaks. Remember that we want to maintain the integrity of our data handling as we process data with Go, and part of that process is ensuring that, when possible, our data is interpretable and readable. JSON turns out to be very useful in achieving these goals (which is why it is also used for logging, in many cases).

Go offers really great JSON functionality in its standard library with `encoding/json`. We will utilize this standard library functionality throughout the book.

Parsing JSON

To understand how to parse (that is, unmarshal) JSON data in Go, we will be using some data from the Citi Bike API (`https://www.citibikenyc.com/system-data`), a bike-sharing service operating in New York City. Citi Bike provides frequently updated operational information about its network of bike sharing stations in JSON format at `https://gbfs.citibikenyc.com/gbfs/en/station_status.json`:

```
{
  "last_updated": 1495252868,
  "ttl": 10,
  "data": {
    "stations": [
      {
        "station_id": "72",
        "num_bikes_available": 10,
        "num_bikes_disabled": 3,
        "num_docks_available": 26,
        "num_docks_disabled": 0,
        "is_installed": 1,
        "is_renting": 1,
        "is_returning": 1,
        "last_reported": 1495249679,
        "eightd_has_available_keys": false
      },
```

```
        {
          "station_id": "79",
          "num_bikes_available": 0,
          "num_bikes_disabled": 0,
          "num_docks_available": 33,
          "num_docks_disabled": 0,
          "is_installed": 1,
          "is_renting": 1,
          "is_returning": 1,
          "last_reported": 1495248017,
          "eightd_has_available_keys": false
        },

        etc...

        {
          "station_id": "3464",
          "num_bikes_available": 1,
          "num_bikes_disabled": 3,
          "num_docks_available": 53,
          "num_docks_disabled": 0,
          "is_installed": 1,
          "is_renting": 1,
          "is_returning": 1,
          "last_reported": 1495250340,
          "eightd_has_available_keys": false
        }
      ]
    }
}
```

To parse the import and this type of data in Go, we first need to import `encoding/json` (along with a couple of other things from a standard library, such as `net/http`, because we are going to pull this data off of the previously mentioned website). We will also define `struct` that mimics the structure of the JSON shown in the preceding code:

```
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

// citiBikeURL provides the station statuses of CitiBike bike sharing
stations.
const citiBikeURL =
"https://gbfs.citibikenyc.com/gbfs/en/station_status.json"
```

```
// stationData is used to unmarshal the JSON document returned form
citiBikeURL.
type stationData struct {
    LastUpdated int `json:"last_updated"`
    TTL int `json:"ttl"`
    Data struct {
        Stations []station `json:"stations"`
    } `json:"data"`
}

// station is used to unmarshal each of the station documents in
stationData.
type station struct {
    ID string `json:"station_id"`
    NumBikesAvailable int `json:"num_bikes_available"`
    NumBikesDisabled int `json:"num_bike_disabled"`
    NumDocksAvailable int `json:"num_docks_available"`
    NumDocksDisabled int `json:"num_docks_disabled"`
    IsInstalled int `json:"is_installed"`
    IsRenting int `json:"is_renting"`
    IsReturning int `json:"is_returning"`
    LastReported int `json:"last_reported"`
    HasAvailableKeys bool `json:"eightd_has_available_keys"`
}
```

Note a couple of things here: (i) we have followed Go idioms by avoiding the `struct` field name with underscores, but (ii) we have utilized the `json` struct tags to label the `struct` fields with the corresponding expected fields in the JSON data.



Note, to properly parse JSON data, the struct fields need to be exported fields. That is, the fields need to begin with a capital letter. `encoding/json` does cannot view fields using reflect unless they are exported.

Now we can get the JSON data from the URL and unmarshal it into a new `stationData` value. This will produce a `struct` variable with the respective fields filled with the data in the tagged JSON data fields. We can check it by printing out some data associated with one of the stations:

```
// Get the JSON response from the URL.
response, err := http.Get(citiBikeURL)
if err != nil {
    log.Fatal(err)
}
defer response.Body.Close()

// Read the body of the response into []byte.
```

```
body, err := ioutil.ReadAll(response.Body)
if err != nil {
    log.Fatal(err)
}

// Declare a variable of type stationData.
var sd stationData

// Unmarshal the JSON data into the variable.
if err := json.Unmarshal(body, &sd); err != nil {
    log.Fatal(err)
}

// Print the first station.
fmt.Printf("%+v\n\n", sd.Data.Stations[0])
```

When we run this, we can see that our `struct` contains the parsed data from the URL:

```
$ go build
$ ./myprogram
{ID:72 NumBikesAvailable:11 NumBikesDisabled:0 NumDocksAvailable:25
NumDocksDisabled:0 IsInstalled:1 IsRenting:1 IsReturning:1
LastReported:1495252934 HasAvailableKeys:false}
```

JSON output

Now let's say that we have the Citi Bike station data in our `stationData` struct value and we want to save that data out to a file. We can do this with `json.marshal`:

```
// Marshal the data.
outputData, err := json.Marshal(sd)
if err != nil {
    log.Fatal(err)
}

// Save the marshalled data to a file.
if err := ioutil.WriteFile("citibike.json", outputData, 0644); err != nil {
    log.Fatal(err)
}
```

SQL-like databases

Although there is a good bit of hype around interesting NoSQL databases and key-value stores, SQL-like databases are still ubiquitous. Every data scientist will, at some point, be processing data from an SQL-like database, such as Postgres, MySQL, or SQLite.

For example, we may be required to query one or more tables in a Postgres database to generate a set of features for model training. After using that model to make predictions or identify anomalies, we may send results to another database table that drives a dashboard or other reporting tool.

Go, of course, interacts nicely with all the popular data stores, such as SQL, NoSQL, key-value, and so on, but here, we will focus on SQL-like interactions. We will utilize `database/sql` for these interactions throughout the book.

Connecting to an SQL database

The first thing we need do before connecting to an SQL-like database is identify the particular database that we will be interacting with and import a corresponding driver. In the following examples, we will be connecting to a Postgres database and will utilize the `github.com/lib/pq` database driver for `database/sql`. This driver can be loaded via an empty import (with a corresponding comment):

```
import (
    "database/sql"
    "fmt"
    "log"
    "os"

    // pq is the library that allows us to connect
    // to postgres with databases/sql.
    _ "github.com/lib/pq"
)
```

Now let's assume that you have exported the Postgres connection string to an environmental variable `PGURL`. We can easily create an `sql.DB` value for our connection via the follow code:

```
// Get the postgres connection URL. I have it stored in
// an environmental variable.
pgURL := os.Getenv("PGURL")
if pgURL == "" {
    log.Fatal("PGURL empty")
}
```

```
// Open a database value. Specify the postgres driver
// for databases/sql.
db, err := sql.Open("postgres", pgURL)
if err != nil {
    log.Fatal(err)
}
defer db.Close()
```

Note that we need to defer the `close` method on this value. Also, note that creating this value does not mean that you have made a successful connection to the database. This is merely a value used by `database/sql` to connect to the database when triggered to do so by certain operations (such as a query).

To ensure that we can make a successful connection to the database, we can use the `Ping` method:

```
if err := db.Ping(); err != nil {
    log.Fatal(err)
}
```

Querying the database

Now that we know how to connect to the database, let's see how we can get data out of the database. We won't cover the specifics of SQL queries and statements in this book. If you are not familiar with SQL, I would highly recommend that you learn how to query, insert, and so on, but for our purposes here, you should know that there are basically two types of operations we want to perform as related to SQL databases:

- A `Query` operation selects, groups, or aggregates data in the database and returns rows of data to us
- An `Exec` operation updates, inserts, or otherwise modifies the state of the database without an expectation that portions of the data stored in the database should be returned

As you might expect, to get data out of our database, we will use a `Query` operation. To do this, we need to query the database with an SQL statement string. For example, imagine we have a database storing a bunch of iris flower measurements (petal length, petal width, and so on), we could query some of that data related to a particular iris species as follows:

```
// Query the database.
rows, err := db.Query(`
    SELECT
        sepal_length as sLength,
```

```
        sepal_width as sWidth,
        petal_length as pLength,
        petal_width as pWidth
    FROM iris
    WHERE species = $1`, "Iris-setosa")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
```

Note that this returns a pointer to an `sql.Rows` value, and we need to defer the closing of this rows value. Then we can loop over our rows and parse the data into values of expected type. We utilize the `Scan` method on rows to parse out the columns returned by the SQL query and print them to standard out:

```
// Iterate over the rows, sending the results to
// standard out.
for rows.Next() {

    var (
        sLength float64
        sWidth float64
        pLength float64
        pWidth float64
    )

    if err := rows.Scan(&sLength, &sWidth, &pLength, &pWidth); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%.2f, %.2f, %.2f, %.2f\n", sLength, sWidth, pLength,
pWidth)
}
```

Finally, we need to check for any errors that might have occurred while processing our rows. We want to maintain the integrity of our data handling, and we cannot assume that we looped over all the rows without encountering an error:

```
// Check for errors after we are done iterating over rows.
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

Modifying the database

As mentioned earlier, there is another flavor of interaction with the database called `Exec`. With these types of statements, we are concerned with updating, adding to, or otherwise modifying the state of one or more tables in the database. We use the same type of database connection, but instead of calling `db.Query`, we will call `db.Exec`.

For example, let's say we want to update some of the values in our iris database table:

```
// Update some values.
res, err := db.Exec("UPDATE iris SET species = 'setosa' WHERE species =
'Iris-setosa'")
if err != nil {
    log.Fatal(err)
}
```

But how do we know whether we were successful and changed something? Well, the `res` function returned here allows us to see how many rows of our table were affected by our update:

```
// See how many rows where updated.
rowCount, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}

// Output the number of rows to standard out.
log.Printf("affected = %d\n", rowCount)
```

Caching

Sometimes, our machine learning algorithms will be trained by and/or given input for prediction via data from external sources (for example, APIs), that is, data that isn't local to the application running our modeling or analysis. Further, we might have various sets of data that are being accessed frequently, may be accessed again soon, or may need to be made available while the application is running.

In at least some of these cases, it might make sense to cache data in memory or embed the data locally where the application is running. For example, if you are reaching out to a government API (typically having high latency) for census data frequently, you may consider maintaining a local or in-memory cache of the census data being used so that you can avoid constantly reaching out to the API.

Caching data in memory

To cache a series of values in memory, we will use `github.com/patrickmn/go-cache`. With this package, we can create an in-memory cache of keys and corresponding values. We can even specify things, such as the time to live, in the cache for specific key-value pairs.

To create a new in-memory cache and set a key-value pair in the cache, we do the following:

```
// Create a cache with a default expiration time of 5 minutes, and which
// purges expired items every 30 seconds
c := cache.New(5*time.Minute, 30*time.Second)

// Put a key and value into the cache.
c.Set("mykey", "myvalue", cache.DefaultExpiration)
```

To then retrieve the value for `mykey` out of the cache, we just need to use the `Get` method:

```
v, found := c.Get("mykey")
if found {
    fmt.Printf("key: mykey, value: %s\n", v)
}
```

Caching data locally on disk

The caching we just saw is in memory. That is, the cached data exists and is accessible while your application is running, but as soon as your application exits, your data disappears. In some cases, you may want your cached data to stick around when your application restarts or exits. You may also want to back up your cache such that you don't have to start applications from scratch without a cache of relevant data.

In these scenarios, you may consider using a local, embedded cache, such as `github.com/boltdb/bolt`. BoltDB, as it is referred to, is a very popular project for these sorts of applications, and basically consists of a local key-value store. To initialize one of these local key-value stores, do the following:

```
// Open an embedded.db data file in your current directory.
// It will be created if it doesn't exist.
db, err := bolt.Open("embedded.db", 0600, nil)
if err != nil {
    log.Fatal(err)
}
defer db.Close()

// Create a "bucket" in the boltdb file for our data.
if err := db.Update(func(tx *bolt.Tx) error {
```

```
        _, err := tx.CreateBucket([]byte("MyBucket"))
        if err != nil {
            return fmt.Errorf("create bucket: %s", err)
        }
        return nil
    }); err != nil {
        log.Fatal(err)
    }
```

You can, of course, have multiple different buckets of data in your BoltDB and use a filename other than `embedded.db`.

Next, let's say you had a map of string values in memory that you need to cache in BoltDB. To do this, you would range over the keys and values in the map, updating your BoltDB:

```
// Put the map keys and values into the BoltDB file.
if err := db.Update(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    err := b.Put([]byte("mykey"), []byte("myvalue"))
    return err
}); err != nil {
    log.Fatal(err)
}
```

Then, to get values out of BoltDB, you can view your data:

```
// Output the keys and values in the embedded
// BoltDB file to standard out.
if err := db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    c := b.Cursor()
    for k, v := c.First(); k != nil; k, v = c.Next() {
        fmt.Printf("key: %s, value: %s\n", k, v)
    }
    return nil
}); err != nil {
    log.Fatal(err)
}
```

Data versioning

As mentioned, machine learning models produce extremely different results depending on the training data you use, the choices of parameters, and the input data. It is essential to be able to reproduce results for collaborative, creative, and compliance reasons:

- **Collaboration**: Despite what you see on social media, there are no data science and machine learning unicorns (that is, people with knowledge and capabilities in every area of data science and machine learning). We need to have our colleagues' reviews and improve on our work, and this is impossible if they aren't able to reproduce our model results and analyses.
- **Creativity**: I don't know about you, but I have trouble remembering even what I did yesterday. We can't trust ourselves to always remember our reasoning and logic, especially when we are dealing with machine learning workflows. We need to track exactly what data we are using, what results we created, and how we created them. This is the only way we will be able to continually improve our models and techniques.
- **Compliance**: Finally, we may not have a choice regarding data versioning and reproducibility in machine learning very soon. Laws are being passed around the world (for example, the **General Data Protection Regulation** (**GDPR**) in the European Union) that give users a right to an explanation for algorithmically made decisions. We simply cannot hope to comply with these rulings if we don't have a robust way of tracking what data we are processing and what results we are producing.

There are multiple open source data versioning projects. Some of these are focused on security and peer-to-peer distributed storage of data. Others are focused on data science workflows. In this book, we will focus on and utilize Pachyderm (`http://pachyderm.io/`), an open source framework for data versioning and data pipelining. Some of the reasons for this will be clear later in the book when we talk about production deploys and managing ML pipelines. For now, I will just summarize some of the features of Pachyderm that make it an attractive choice for data versioning in Go-based (and other) ML projects:

- It has an convenient Go client, `github.com/pachyderm/pachyderm/src/client`
- The ability to version any type and format of data
- A flexible object store backing for the versioned data
- Integration with a data pipelining system for driving versioned ML workflows

Pachyderm jargon

Think about versioning data in Pachyderm kind of like versioning code in Git. The primitives are similar:

- **Repositories**: These are versioned collections of data, similar to having versioned collections of code in Git repositories
- **Commits**: Data is versioned in Pachyderm by making commits of that data into data repositories
- **Branches**: These lightweight points to certain commits or sets of commits (for example, master points to the latest HEAD commit)
- **Files**: Data is versioned at the file level in Pachyderm, and Pachyderm automatically employs strategies, such as de-duplication, to keep your versioned data space efficient



Even though versioning data with Pachyderm feels similar to versioning code with Git, there are some major differences. For example, merging data doesn't exactly make sense. If there are merge conflicts on petabytes of data, no human could resolve these. Furthermore, the Git protocol would not be space efficient in general for large sets of data. Pachyderm uses its own internal logic to perform the versioning and work with versioned data, and the logic is both space efficient and processing efficient in terms of caching.

Deploying/installing Pachyderm

We will be using Pachyderm in various other places in the book to both version data and create distributed ML workflows. Pachyderm itself is an app that runs on top of Kubernetes (`https://kubernetes.io/`), and is backed by an object store of your choice. For the purposes of this book, development, and experimentation, you can easily install and run Pachyderm locally. It should take 5-10 minutes to install and doesn't require much effort. The instructions for the local installation can be found in the Pachyderm documentation at `http://docs.pachyderm.io`.

When you are ready to run your workflows in production or your deploy model, you can easily deploy a production-ready Pachyderm cluster that will behave the same exact way as your local installation. Pachyderm can be deployed to any cloud, or even on premises.

As mentioned, Pachyderm is an open source project and has an active group of users. If you have questions or need help, you can join the public Pachyderm Slack channel by visiting `http://slack.pachyderm.io/`. The active Pachyderm users and the Pachyderm team itself will be able to respond very quickly to your questions there.

Creating data repositories for data versioning

If you followed the local installation of Pachyderm specified in the Pachyderm documentation, you should have the following:

- Kubernetes running in a Minikube VM on your machine
- The `pachctl` command line tool installed and connected to your Pachyderm cluster

Of course, if you have a production cluster running in a cloud, the following steps still apply. Your `pachctl` would just be connected to the remote cluster.



We will be demonstrating data versioning functionality with the `pachctl` **Command-line Interface** (**CLI**) tool below (which is a Go program). However, as mentioned above, Pachyderm has a full-fledged Go client. You can create repositories, commit data, and much more directly from your Go programs. This functionality will be demonstrated later in Chapter 9, *Deploying and distributing Analyses and Models*.

To create a repository of data called `myrepo`, you can run this code:

```
$ pachctl create-repo myrepo
```

You can then confirm that the repository exists with `list-repo`:

```
$ pachctl list-repo
NAME CREATED SIZE
myrepo 2 seconds ago 0 B
```

This `myrepo` repository is a collection of data that we have defined and is ready for housing-versioned data. Right now, there is no data in the repository, because we haven't put any data there yet.

Putting data into data repositories

Let's say that we have a simple text file:

```
$ cat blah.txt
This is an example file.
```

If this file is part of the data we are utilizing in our ML workflow, we should version it. To version this file in our repository, `myrepo`, we just need to commit it into that repository:

```
$ pachctl put-file myrepo master -c -f blah.txt
```

The `-c` flag specifies that we want Pachyderm to open a new commit, insert the file we are referencing, and close the commit all in one shot. The `-f` flag specifies that we are providing a file.

Note that we are committing a single file to the master branch of a single repository here. However, the Pachyderm API is incredibly flexible. We can commit, delete, or otherwise modify many versioned files in a single commit or over multiple commits. Further, these files could be versioned via a URL, object store link, database dump, and so on.

As a sanity check, we can confirm that our file was versioned in the repository:

```
$ pachctl list-repo
NAME CREATED SIZE
myrepo 10 minutes ago 25 B
$ pachctl list-file myrepo master
NAME TYPE SIZE
blah.txt file 25 B
```

Getting data out of versioned data repositories

Now that we have versioned data in Pachyderm, we probably want to know how to interact with that data. The primary way is via Pachyderm data pipelines (which will be discussed later in this book). The mechanism for interacting with versioned data when using pipelines is a simple file I/O.

However, if we manually want to pull certain sets of versioned data out of Pachyderm, analyze them interactively, then we can use the `pachctl` CLI to get data:

```
$ pachctl get-file myrepo master blah.txt
This is an example file.
```

References

CSV data:

- `encoding/csv` docs: `https://golang.org/pkg/encoding/csv/`
- `github.com/kniren/gota/dataframe` docs: `https://godoc.org/github.com/kniren/gota/dataframe`

JSON data:

- `encoding/json` docs: `https://golang.org/pkg/encoding/json/`
- Bill Kennedy's blog post of JSON decoding: `https://www.goinggo.net/2014/01/decode-json-documents-in-go.html`
- Ben Johnson's blog post Go Walkthrough: `encoding/json` package: `https://medium.com/go-walkthrough/go-walkthrough-encoding-json-package-9681d1d37a8f`

Caching:

- `github.com/patrickmn/go-cache` docs: `https://godoc.org/github.com/patrickmn/go-cache`
- `github.com/boltdb/bolt` docs: `https://godoc.org/github.com/boltdb/bolt`
- Information about and motivation for BoltDB: `https://npf.io/2014/07/intro-to-boltdb-painless-performant-persistence/`

Pachyderm:

- General documentation: `http://docs.pachyderm.io`
- Go client docs: `https://godoc.org/github.com/pachyderm/pachyderm/src/client`
- Public users Slack team registration: `http://docs.pachyderm.io`

Summary

In this chapter, you learned how to gather, organize, and parse data. This is the first step, and one of the most important step, in developing machine learning models, but having data does not get us very far if we do not gain some intuition about our data and put it into a standard form for processing. Next, we will tackle some techniques for further structuring our data (matrices) and for understanding our data (statistics and probability).

2
Matrices, Probability, and Statistics

Although we will take a mostly practical/applied approach to machine learning throughout this book, certain fundamental topics are essential to understand and properly apply machine learning. In particular, a fundamental understanding of probability and statistics will allow us to match certain algorithms with relevant problems, understand our data and results, and apply necessary transformations to our data. Matrices and a little linear algebra will then allow us to properly represent our data and implement optimizations, minimizations, and matrix-based transformations.

Do not worry too much if you are a little rusty in math or statistics. We will cover a few of the basics here and show you how to programmatically work with the relevant statistical measures and matrix techniques that will be utilized later in the book. That being said, this is not a book on statistics, probability, and linear algebra. To truly be proficient in machine learning, one should take time to learn these subjects on a deeper level.

Matrices and vectors

If you spend much time learning and applying machine learning, you will see a bunch of references to matrices and vectors. In fact, many machine learning algorithms boil down to a series of iterative operations on matrices. What are matrices and vectors, and how do we represent them in our Go programs?

For the most part, we will utilize packages from `github.com/gonum` to form and work with matrices and vectors. This is a great series of Go packages focused on numerical computing, and they just keep getting better and better.

Vectors

A vector is an ordered collection of numbers arranged in either a row (left to right) or column (up and down). Each of the numbers in a vector is called a component. This might be, for example, a collection of numbers that represents our company sales, or it might be a collection of numbers representing temperatures.

It's, of course, natural for us to use Go slices to represent these ordered collections of data, as follows:

```
// Initialize a "vector" via a slice.
var myvector []float64

// Add a couple of components to the vector.
myvector = append(myvector, 11.0)
myvector = append(myvector, 5.2)

// Output the results to stdout.
fmt.Println(myvector)
```

Slices are indeed ordered collections. However, they don't really represent the concept of rows or columns, and we would still need to work out various vector operations on top of slices. Thankfully, on the vector operation side, gonum provides `gonum.org/v1/gonum/floats` to operate on slices of `float64` values and `gonum.org/v1/gonum/mat`, which, along with matrices, provides a `Vector` type (with corresponding methods):

```
// Create a new vector value.
myvector := mat.NewVector(2, []float64{11.0, 5.2})
```

Vector operations

As mentioned here, working with vectors necessitates the use of certain vector-/matrix-specific operations and rules. For example, how do we multiply vectors together? How do we know if two vectors are similar? Both `gonum.org/v1/gonum/floats` and `gonum.org/v1/gonum/mat` provide built-in methods and functions for vector/slice operations, such as dot products, sorting, and distance. We won't cover all of the functionality here, as there is quite a bit, but we can get a general feel for how we might work with vectors. First, we can work with `gonum.org/v1/gonum/floats` in the following way:

```
// Initialize a couple of "vectors" represented as slices.
vectorA := []float64{11.0, 5.2, -1.3}
```

```
vectorB := []float64{-7.2, 4.2, 5.1}

// Compute the dot product of A and B
// (https://en.wikipedia.org/wiki/Dot_product).
dotProduct := floats.Dot(vectorA, vectorB)
fmt.Printf("The dot product of A and B is: %0.2f\n", dotProduct)

// Scale each element of A by 1.5.
floats.Scale(1.5, vectorA)
fmt.Printf("Scaling A by 1.5 gives: %v\n", vectorA)

// Compute the norm/length of B.
normB := floats.Norm(vectorB, 2)
fmt.Printf("The norm/length of B is: %0.2f\n", normB)
```

We can also do similar operations with `gonum.org/v1/gonum/mat`:

```
// Initialize a couple of "vectors" represented as slices.
vectorA := mat.NewVector(3, []float64{11.0, 5.2, -1.3})
vectorB := mat.NewVector(3, []float64{-7.2, 4.2, 5.1})

// Compute the dot product of A and B
// (https://en.wikipedia.org/wiki/Dot_product).
dotProduct := mat.Dot(vectorA, vectorB)
fmt.Printf("The dot product of A and B is: %0.2f\n", dotProduct)

// Scale each element of A by 1.5.
vectorA.ScaleVec(1.5, vectorA)
fmt.Printf("Scaling A by 1.5 gives: %v\n", vectorA)

// Compute the norm/length of B.
normB := blas64.Nrm2(3, vectorB.RawVector())
fmt.Printf("The norm/length of B is: %0.2f\n", normB)
```

The semantics are similar in the two cases. If you are only working with vectors (not matrices), and/or you just need some lightweight and quick operations on slices of floats, then `gonum.org/v1/gonum/floats` is likely a good choice. However, if you are working with both matrices and vectors, and/or want access to a wider range of vector/matrix functionality, you are likely better off with `gonum.org/v1/gonum/mat` (along with occasional references to `gonum.org/v1/gonum/blas/blas64`).

Matrices

Matrices and linear algebra may seem complicated to many people, but simply put, matrices are just rectangular organizations of numbers, and linear algebra dictates the rules associated with their manipulation. For example, a matrix A with numbers arranged on a 4 x 3 rectangle may look like this:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

The components of A (a_{11}, a_{12}, and so on) are the individual numbers that we are arranging into a matrix, and the subscripts indicate the location of the components within the matrix. The first index is the row index and the second index is the column index. More generally, A could have any shape/size with M rows and N columns:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{pmatrix}$$

To form a matrix like this with `gonum.org/v1/gonum/mat`, we need to create a slice of `float64` values that is a flat representation of all the matrix components. For instance, in our example, we want to form the following matrix:

$$\begin{pmatrix} 1.2 & -5.7 \\ -2.4 & 7.3 \end{pmatrix}$$

We would need to create a slice of `float64` values as follows:

```
// Create a flat representation of our matrix.
components := []float64{1.2, -5.7, -2.4, 7.3}
```

Then, we can supply this, along with dimension information, to `gonum.org/v1/gonum/mat` to form a new `mat.Dense` matrix value:

```
// Form our matrix (the first argument is the number of
// rows and the second argument is the number of columns).
a := mat.NewDense(2, 2, data)

// As a sanity check, output the matrix to standard out.
fa := mat.Formatted(a, mat.Prefix(" "))
fmt.Printf("mat = %v\n\n", fa)
```

Note that we have also used the nice formatting logic in `gonum.org/v1/gonum/mat` to print the matrix as a sanity check. When you run this, you should see the following:

```
$ go build
$ ./myprogram
A = [ 1.2 -5.7]
    [-2.4  7.3]
```

We can then access and modify certain values within *A* via built-in methods:

```
// Get a single value from the matrix.
val := a.At(0, 1)
fmt.Printf("The value of a at (0,1) is: %.2f\n\n", val)

// Get the values in a specific column.
col := mat.Col(nil, 0, a)
fmt.Printf("The values in the 1st column are: %v\n\n", col)

// Get the values in a kspecific row.
row := mat.Row(nil, 1, a)
fmt.Printf("The values in the 2nd row are: %v\n\n", row)

// Modify a single element.
a.Set(0, 1, 11.2)

// Modify an entire row.
a.SetRow(0, []float64{14.3, -4.2})

// Modify an entire column.
a.SetCol(0, []float64{1.7, -0.3})
```

Matrix operations

As with vectors, matrices have their own set of rules for arithmetic, along with a whole set of special operations. Some of the arithmetic associated with matrices behaves in a similar way to what you might expect. However, you need to take special care when doing things such as multiplying matrices together or taking an inverse.

Conveniently, `gonum.org/v1/gonum/mat` provides a nice API for this arithmetic and many other special operations. Here is an example showing a few operations, such as adding, multiplying, dividing, and so on:

```
// Create two matrices of the same size, a and b.
a := mat.NewDense(3, 3, []float64{1, 2, 3, 0, 4, 5, 0, 0, 6})
b := mat.NewDense(3, 3, []float64{8, 9, 10, 1, 4, 2, 9, 0, 2})

// Create a third matrix of a different size.
c := mat.NewDense(3, 2, []float64{3, 2, 1, 4, 0, 8})

// Add a and b.
d := mat.NewDense(0, 0, nil)
d.Add(a, b)
fd := mat.Formatted(d, mat.Prefix("            "))
fmt.Printf("d = a + b = %0.4v\n\n", fd)

// Multiply a and c.
f := mat.NewDense(0, 0, nil)
f.Mul(a, c)
ff := mat.Formatted(f, mat.Prefix("          "))
fmt.Printf("f = a c = %0.4v\n\n", ff)

// Raising a matrix to a power.
g := mat.NewDense(0, 0, nil)
g.Pow(a, 5)
fg := mat.Formatted(g, mat.Prefix("          "))
fmt.Printf("g = a^5 = %0.4v\n\n", fg)

// Apply a function to each of the elements of a.
h := mat.NewDense(0, 0, nil)
sqrt := func(_, _ int, v float64) float64 { return math.Sqrt(v) }
h.Apply(sqrt, a)
fh := mat.Formatted(h, mat.Prefix("              "))
fmt.Printf("h = sqrt(a) = %0.4v\n\n", fh)
```

In particular, note the `Apply()` method above. This functionality is extremely useful as it allow you to apply any function to the elements of a matrix. You can apply the same function to all elements or make the function dependent on the indices of the matrix elements. For example, you could you this method to perform element-wise multiplications, applications of user defined functions, or applications of functions from third party packages.

Then, for all the various things, such as determinants, eigenvalue/vector solvers, and inverses, `gonum.org/v1/gonum/mat` has you covered. Again, I won't expand on all of the functionality, but here is a sample of some of the operations:

```
// Create a new matrix a.
a := mat.NewDense(3, 3, []float64{1, 2, 3, 0, 4, 5, 0, 0, 6})

// Compute and output the transpose of the matrix.
ft := mat.Formatted(a.T(), mat.Prefix(" "))
fmt.Printf("a^T = %v\n\n", ft)

// Compute and output the determinant of a.
deta := mat.Det(a)
fmt.Printf("det(a) = %.2f\n\n", deta)

// Compute and output the inverse of a.
aInverse := mat.NewDense(0, 0, nil)
if err := aInverse.Inverse(a); err != nil {
    log.Fatal(err)
}
fi := mat.Formatted(aInverse, mat.Prefix(" "))
fmt.Printf("a^-1 = %v\n\n", fi)
```

Note that in this example, we leverage Go's explicit error handling functionality when we need to ensure that we are maintaining integrity and readability. Matrices don't always have inverses. There are various situations like this that arise when working with matrices and large datasets, and we want to ensure that our application behaves as expected.

Statistics

At the end of the day, the success of your machine learning application is going to come down to the quality of your data, your understanding of the data, and your evaluation/validation of the results. All three of these things require us to have an understanding of statistics.

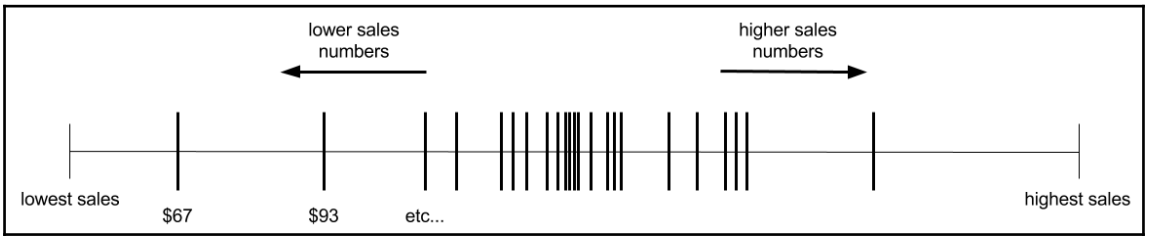
The field of statistics helps us to gain an understanding of our data, and to quantify what our data and results look like. It also provides us with mechanisms to measure how well our application is performing and prevent certain machine learning pitfalls (such as overfitting).

As with linear algebra, we aren't able to give a complete introduction to statistics here, but there are many resources online and in print to learn introductory statistics. Here we will focus on a fundamental understanding of the basics, along with the practicalities of implementation in Go. We will introduce the idea of distributions, along with an introduction to quantifying and visualizing these distributions.

Distributions

A distribution is a representation of how often values appear within a dataset. Let's say, for instance, that one thing you are tracking as a data scientist is the daily sales of a certain product or service, and you have a long list (which you could represent as a vector or part of a matrix) of these daily sales numbers. These sales numbers are part of our dataset, and they include one day with sales of $121, another day with sales of $207, and so on.

There will be one sales number that is the lowest out of the one we have accumulated. There will also be one sales number that is the highest out of the one we have accumulated, and the rest of the sales numbers that are somewhere in between (at least if we assume no exact duplicates). The following image represents these low, high, and in-between values of sales along a line:



This is, thus, a distribution of sales, or at least one representation of the distribution of sales. Note that this distribution has areas where there are more numbers and areas where the numbers are a little sparse. Additionally, note that there seems to be a tendency for numbers to be near the center of the distribution.

Statistical measures

To quantify what a distribution looks like, we will use various statistical measures. Generally, there are two types of these measures:

1. **Central tendency measures**: These measure where most of the values are located, or where the *center* of the distribution is located (for example, along the preceding linear representation).
2. **Spread or dispersion measures**: These measure how the values of the distribution are spread across the distribution's range (from the lowest value to the highest value).

There are various packages that allow you to quickly calculate and/or utilize these statistical measures. We will make use of `gonum.org/v1/gonum/stat` (you are probably starting to notice that we will be making heavy use of gonum) and `github.com/montanaflynn/stats`.



Note, there is a one letter difference in the names of the `gonum.org/v1/gonum/stat` and `github.com/montanaflynn/stats` packages. Keep this in mind as you review the examples in the following sections.

Measures of central tendency

Measures of central tendency include the following:

- `Mean`: This is what you might commonly refer to as an average. We calculate this by summing all of the numbers in the distribution and then dividing by the count of the numbers.
- `Median`: If we sort all of the numbers in our distribution from the lowest to highest, this is the number that separates the lowest half of the numbers from the highest half of the numbers.
- `Mode`: This is the most frequently occurring value in the distribution.

Let's calculate these measures for the values in one column of the iris dataset previously introduced in `Chapter 1`, *Gathering and Organizing Data*. As a reminder, this dataset included four columns of flower measurements, along with a column of the corresponding flower species. Thus, each of the measurement columns includes a bunch of values that represent a distribution of that measurement:

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Get the float values from the "sepal_length" column as
// we will be looking at the measures for this variable.
sepalLength := irisDF.Col("sepal_length").Float()

// Calculate the Mean of the variable.
meanVal := stat.Mean(sepalLength, nil)

// Calculate the Mode of the variable.
modeVal, modeCount := stat.Mode(sepalLength, nil)

// Calculate the Median of the variable.
medianVal, err := stats.Median(sepalLength)
if err != nil {
    log.Fatal(err)
}

// Output the results to standard out.
fmt.Printf("\nSepal Length Summary Statistics:\n")
fmt.Printf("Mean value: %0.2f\n", meanVal)
fmt.Printf("Mode value: %0.2f\n", modeVal)
fmt.Printf("Mode count: %d\n", int(modeCount))
fmt.Printf("Median value: %0.2f\n\n", medianVal)
```

Running this program results in the following:

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Mean value: 5.84
Mode value: 5.00
Mode count: 10
```

```
Median value: 5.80
```

You can see that the mean, mode, and median are all slightly different. However, note that the mean and median are very close for the values in the `sepal_length` column.

On the other hand, if we change `sepal_length` to `petal_length` in the preceding code, we will get the following results:

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Mean value: 3.76
Mode value: 1.50
Mode count: 14
Median value: 4.35
```

For the `petal_length` values, the mean and median are not as close. We can already start to get some intuition about the data from this information. If the mean and median aren't close, this implies that high or low values are dragging the mean higher or lower, respectively--an influence that isn't as noticeable in the median. We call this a **skewed distribution**.

Measures of spread or dispersion

Now that we have an idea about where most of our values are located (or the center of our distribution), let's try to quantify how the values of our distribution are spread around the center of our distribution. Some of the widely used measures that quantify this are as follows:

- **Maximum**: The highest value of the distribution
- **Minimum**: The lowest value of the distribution
- **Range**: The difference between the maximum and minimum
- **Variance**: This measure is calculated by taking each of the values in the distribution, calculating each one's difference from the distribution's mean, squaring this difference, adding it to the other squared differences, and dividing by the number of values in the distribution
- **Standard deviation**: The square root of the variance
- **Quantiles/quartiles**: Similar to the median, these measures define cut-off points in the distribution where a certain number of lower values are below the measure and a certain number of higher values are above the measure

Using `gonum.org/v1/gonum/stat`, the calculation of these measures looks as follows:

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Get the float values from the "sepal_length" column as
// we will be looking at the measures for this variable.
sepalLength := irisDF.Col("petal_length").Float()

// Calculate the Max of the variable.
minVal := floats.Min(sepalLength)

// Calculate the Max of the variable.
maxVal := floats.Max(sepalLength)

// Calculate the Median of the variable.
rangeVal := maxVal - minVal

// Calculate the variance of the variable.
varianceVal := stat.Variance(sepalLength, nil)

// Calculate the standard deviation of the variable.
stdDevVal := stat.StdDev(sepalLength, nil)

// Sort the values.
inds := make([]int, len(sepalLength))
floats.Argsort(sepalLength, inds)

// Get the Quantiles.
quant25 := stat.Quantile(0.25, stat.Empirical, sepalLength, nil)
quant50 := stat.Quantile(0.50, stat.Empirical, sepalLength, nil)
quant75 := stat.Quantile(0.75, stat.Empirical, sepalLength, nil)

// Output the results to standard out.
fmt.Printf("\nSepal Length Summary Statistics:\n")
fmt.Printf("Max value: %0.2f\n", maxVal)
fmt.Printf("Min value: %0.2f\n", minVal)
fmt.Printf("Range value: %0.2f\n", rangeVal)
fmt.Printf("Variance value: %0.2f\n", varianceVal)
fmt.Printf("Std Dev value: %0.2f\n", stdDevVal)
fmt.Printf("25 Quantile: %0.2f\n", quant25)
```

```
fmt.Printf("50 Quantile: %0.2f\n", quant50)
fmt.Printf("75 Quantile: %0.2f\n\n", quant75)
```

Running this program gives the following result:

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Max value: 6.90
Min value: 1.00
Range value: 5.90
Variance value: 3.11
Std Dev value: 1.76
25 Quantile: 1.60
50 Quantile: 4.30
75 Quantile: 5.10
```

Okay, let's try to get through these numbers and see what they imply about the distribution of values in the `sepal_length` column. We can deduce the following.

First, the standard deviation is `1.76` and the whole range of values is `5.90`. As opposed to the variance, the standard deviation has the same units as the values themselves, and thus we can see that the values actually vary quite a bit within the range of values (the standard deviation value is about 30% of the total range of values).

Next, let's look at the quantiles. The 25% quantile represents a point in the distribution where 25% of the values in the distribution are below the measure and the other 75% are above. This is similar for the 50% and 75% quantiles. As the 25% quantile is much closer to the minimum than the distance between the 75% quantile and the maximum, we can deduce that the higher values in the distribution are likely more spread out than the lower values.

Of course, you can utilize any combination of these measures, along with the central tendency measures, to help you quantify how a distribution looks, and there are other statistical measures that can't be covered here.



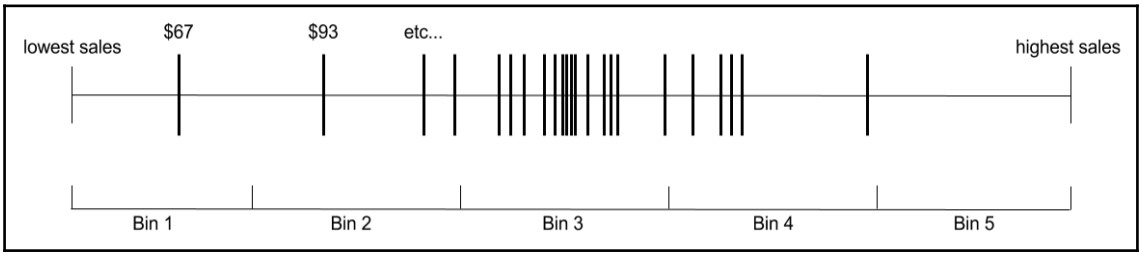
The point here is that you should be making use of measures like these to help you build mental models of your data. This will allow you to put results in context and sanity-check your work.

Visualizing distributions

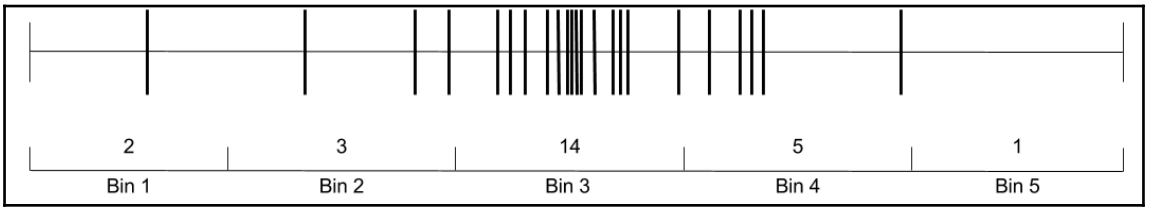
Even though it is important to quantify how a distribution looks, we should actually visualize the distribution to gain the most intuition. There are various types of plots and graphs that allow us to create a visual representation of a distribution of values. These help us form a mental model of the data and communicate information about our data to other members of our team, users of our applications, and so on.

Histograms

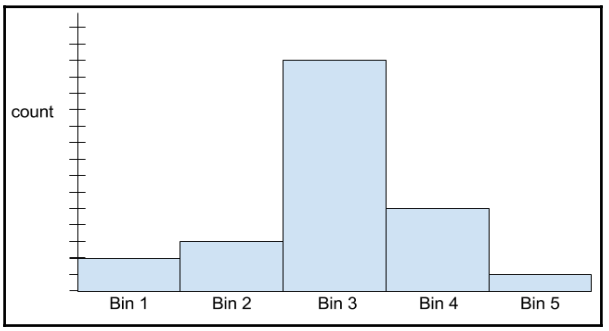
The first type of these graphs or charts that help us understand our distributions is called a **histogram**. Actually, a histogram is really a certain way of organizing or counting your values, which can then be plotted in a histogram plot. To form a histogram, we first create a certain number of bins, carving out different areas of the total range of our values. For example, take the distribution of sales numbers that we discussed in the previous sections:



Next, we count how many of our values are in each of these bins:



These counts, along with the definition of the bins, form our histogram. We can then easily convert this to a plot of the counts, which provides a nice visual representation of our distribution:



We can again use gonum to create a histogram from actual data and plot the histogram. The packages that gonum provides for this type of plotting, along with other types of plotting, can be found in `gonum.org/v1/plot`. As an example, let's create a histogram plot for each of the columns in the iris dataset.

First, import the following from gonum:

```
import (
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
)
```

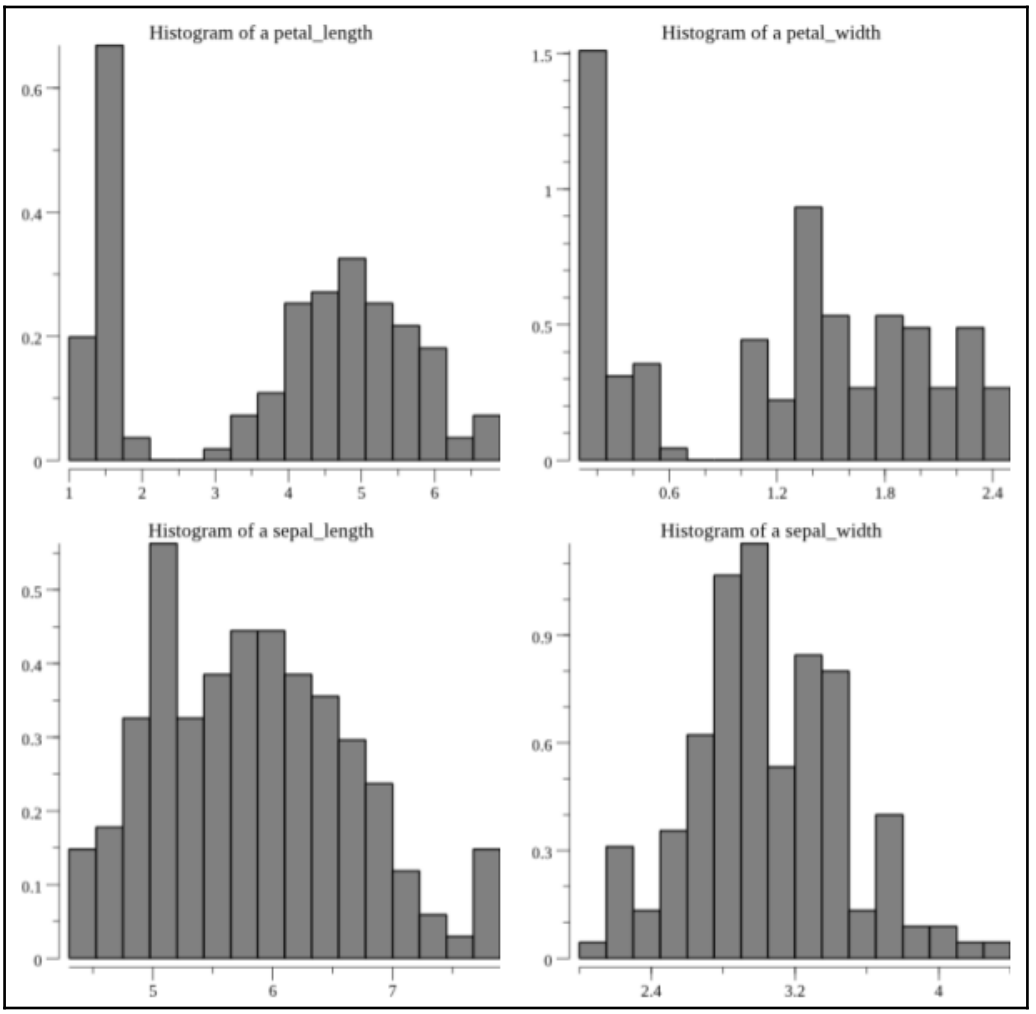
Then we will read in the iris dataset, create a dataframe, and look over the numerical columns generating the histogram plots:

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Create a histogram for each of the feature columns in the dataset.
for _, colName := range irisDF.Names() {
```

```
        // If the column is one of the feature columns, let's create
        // a histogram of the values.
        if colName != "species" {

            // Create a plotter.Values value and fill it with the
            // values from the respective column of the dataframe.
            v := make(plotter.Values, irisDF.Nrow())
            for i, floatVal := range irisDF.Col(colName).Float() {
                v[i] = floatVal
            }

            // Make a plot and set its title.
            p, err := plot.New()
            if err != nil {
                log.Fatal(err)
            }
            p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

            // Create a histogram of our values drawn
            // from the standard normal.
            h, err := plotter.NewHist(v, 16)
            if err != nil {
                log.Fatal(err)
            }

            // Normalize the histogram.
            h.Normalize(1)

            // Add the histogram to the plot.
            p.Add(h)

            // Save the plot to a PNG file.
            if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err !=
    nil {
                log.Fatal(err)
            }
        }
    }
```

Note that we have normalized our histograms (with `h.Normalize()`). This is typical because often you will want to compare different distributions with different counts of values. Normalizing the histogram allows us to compare the different distributions side by side.

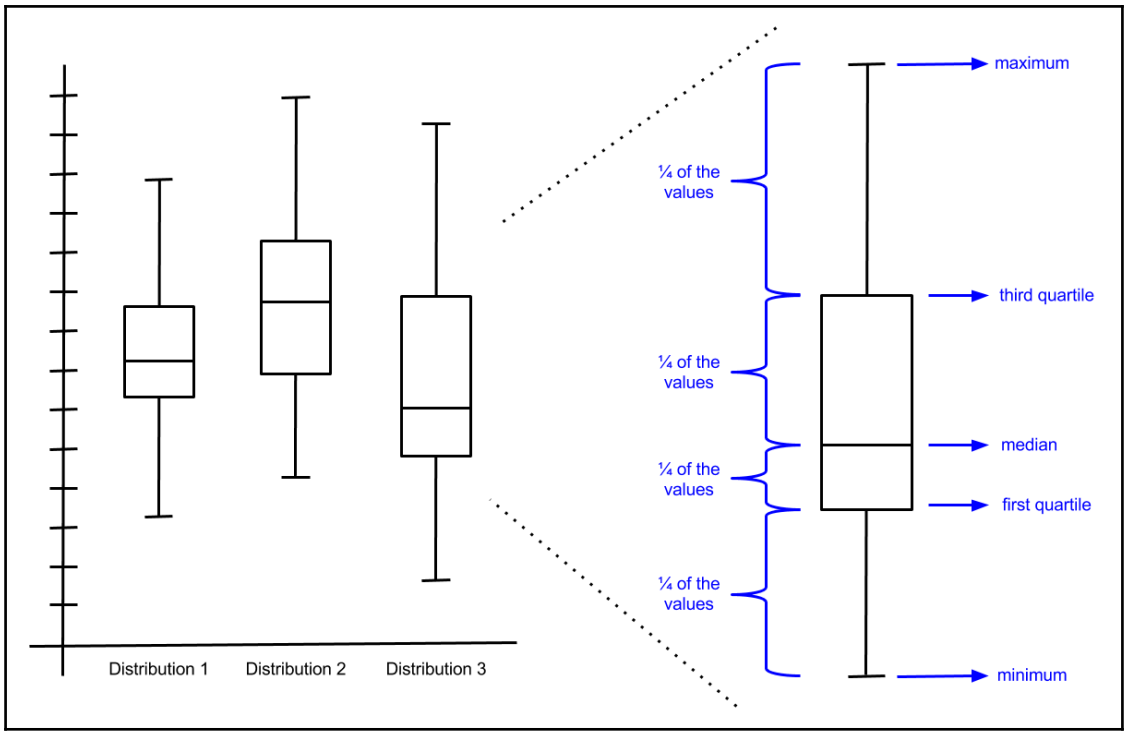
The preceding code will generate four `*.png` files with the following histograms for the numerical columns in the iris dataset:



Each of these distributions looks different from the others. The `sepal_width` distribution looks like a bell curve or normal/Gaussian distribution (which we will discuss later in the book). On the other hand, the petal distributions look like they have two different distinct clumps of values. We will make use of these observations later on when we are developing our machine learning workflows, but for now, just note how these visualizations are able to help us develop a mental model of our data.

Box plots

Histograms are by no means the only way to visually gain an understanding of our data. Another commonly used type of plot is called a **box plot**. This type of plot also gives us an idea about the grouping and spread of values in a distribution, but, as opposed to the histogram, the box plot has several marked features that help guide our eyes:



Because the borders of the boxes in the box plots are defined by the median, **first quartile** (25% quantile/percentile), and **third quartile**, the same number of distribution values that are contained in the two central boxes. If one box is bigger than the other, it means that the distribution is skewed.

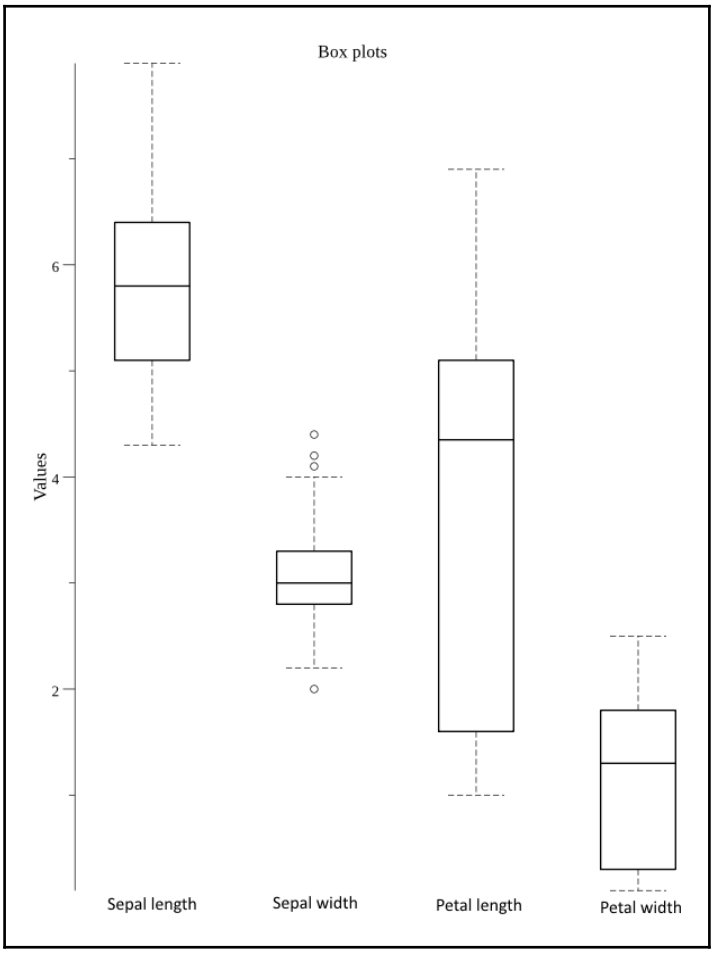
The box plots also include two tails or whiskers. These give us a quick visual indication of the range of the distribution as compared to the area that includes most of the values (the middle 50%).

To solidify this type of plot, let's again create plots for the iris dataset. Similar to the histograms, we will use `gonum.org/v1/plot`. However, in this case, we will put all the box plots into the same `*.png`:

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Create the plot and set its title and axis label.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}

p.Title.Text = "Box plots"
p.Y.Label.Text = "Values"

// Create the box for our data.
w := vg.Points(50)

// Create a box plot for each of the feature columns in the dataset.
for idx, colName := range irisDF.Names() {

    // If the column is one of the feature columns, let's create
    // a histogram of the values.
    if colName != "species" {

        // Create a plotter.Values value and fill it with the
        // values from the respective column of the dataframe.
        v := make(plotter.Values, irisDF.Nrow())
        for i, floatVal := range irisDF.Col(colName).Float() {
            v[i] = floatVal
        }

        // Add the data to the plot.
        b, err := plotter.NewBoxPlot(w, float64(idx), v)
        if err != nil {
            log.Fatal(err)
        }
        p.Add(b)
    }
}
```

```
// Set the X axis of the plot to nominal with
// the given names for x=0, x=1, etc.
p.NominalX("sepal_length", "sepal_width", "petal_length", "petal_width")

if err := p.Save(6*vg.Inch, 8*vg.Inch, "boxplots.png"); err != nil {
    log.Fatal(err)
}
```

This will generate the following figure:



As we observed in the histograms, the `sepal_length` column appears to be relatively symmetrical. On the other hand, `petal_length` appears to be much less symmetrical. Note also that gonum has included several outliers (marked as circles or dots) in the box plots. Many plotting packages include these. They indicate the values that are at least a certain distance away from the median of the distribution.

Probability

At this point, we now understand a couple of ways to represent/manipulate our data (matrices and vectors), and we know how to gain and understanding about our data, and how to quantify how our data looks (statistics). However, sometimes when we are developing machine learning applications, we also want to know how likely it is that a prediction is correct or how significant certain results are, given a history of results. Probability can help us answer these *how likely* and *how significant* questions.

Generally, probability has to do with the likelihood of events or observations. For example, if we are going to flip a coin to make a decision, how likely is it that we would see heads (50%), how likely is it that we would see tails (50%), or even how likely is it that the coin is a fair coin? This might seem like a trivial example, but many similar questions come up when doing machine learning. In fact, some machine learning algorithms are built on probabilistic rules and theorems.

Random variables

Let's say that we have an experiment, like our scenario of flipping the coin, which could have multiple outcomes (heads or tails). Now let's define a variable whose value could be one of these outcomes. This variable is referred to as a **random variable**.

In the case of the coin toss (at least if we are considering a fair coin), each of the outcomes of the random variable is equally likely. That is, we have a 50% chance of seeing a heads outcome and a 50% chance of seeing a tails outcome. Yet, the various values of the random variable need not be equally likely. If we are trying to predict whether it will rain or not, those outcomes will not be equally likely.

Random variables allow us to define these how likely and how significant sort of questions that we mentioned earlier. They can have a finite number of outcomes and/or could represent ranges of continuous variables.

Probability measures

So how likely is it that we observe a specific outcome of an experiment? Well, to quantify the answer to this question, we introduce **probability measures**, which are often referred to as probabilities. They are represented by a number between 0 and 1 and/or by a percentage between 0% and 100%.

In the case of flipping a fair coin, we have the following scenario:

- There is a 0.5 or 50% chance, or probability, of heads, where 0.5 or 50% is a probability measure
- There is a 0.5 or 50% chance of tails

Probability measures for a certain experiment must add up to 1 because when an event occurs, it has to correspond to one of the possible outcomes.

Independent and conditional probability

Two events (outcomes of experiments) are independent if the probability of one of the events in no way affects the probability of the other events. An example of independent events is the flipping of coins or the rolling of dice. On the other hand, dependent events are those events where the probability of one event influences the probability of another event. An example of dependent events is drawing cards from a deck of cards without replacement.

How can we quantify this second type of probability, which is commonly referred to as conditional probability? Symbolically, independent probabilities can be represented by $P(A)$, which is the probability of A (where A could represent flipping a coin, rolling a die, and so on). Then conditional probabilities are represented by $P(B|A)$, which is the probability of A given B (where B is another outcome).

To actually calculate a conditional probability, we can use the Bayes theorem/rule: $P(A|B) = P(B|A)\,P(A) / P(B)$. Sometimes you will see these terms defined as follows:

- $P(A|B)$: The *posterior*, because it is something we know about A after observing B
- $P(A)$: The *prior*, because it is data we have about A before observing B
- $P(B|A)$: The *likelihood*, because it is a measure of the compatibility of B with A
- $P(B)$: The *evidence*, because it measures the probability of B, which we already know is true

This theorem is the basis for various techniques that will be discussed later in the book, such as the naive the Bayes technique for classification.

Hypothesis testing

We can quantify *how likely* questions with probabilities and can even calculate conditional probabilities with Bayes theorem, but how can we quantify *how significant* questions that correspond to real-world observations? For example, we can quantify the probability of heads/tails with a fair coin, but how can we determine *how significant* it is when we flip a coin a bunch of times and observe 48% heads and 52% tails? Is this significant? Does it mean that we have an unfair coin?

These *how significant* questions can be answered using a process called hypothesis testing. This process generally includes the following steps:

1. Formulate a **null hypothesis**, referred to as H_0, and an **alternate hypothesis**, referred to as H_a. H_0 represents the scenario where what you observe (for example, 48% heads and 52% tails) is the result of pure chance, whereas H_a represents the scenario where some type of underlying effect is causing significant deviation from pure chance (for example, an unfair coin). The null hypothesis is always assumed to be true.
2. Determine a **test statistic** that you will use to determine the validity of H_0.
3. Determine a **p-value**, which indicates the probability of observing a test statistic at least as significant as your test statistic assuming that H_0 is true. This p-value can be obtained from a probability distribution corresponding to the test statistic (often represented as a table or distribution function).
4. Compare your p-value with a predetermined threshold. If the p-value is less or equal to the predetermined threshold, H_0 is ruled out in favor of H_a.

This might seem rather abstract, but this process will intersect with your machine learning workflows at some point. For example, you may change one of your machine learning models that optimizes advertisements, and then you may want to quantify if an increase in sales is actually statistically significant. In another scenario, you may be analyzing logs representing possible fraudulent network traffic, and you might need to build a model that recognizes statistically significant deviations from expected network traffic.



Note that you may also see certain hypothesis tests referred to as A/B tests, and, although the process listed here is common, it is by no means the only methodology for hypothesis testing. There are methods for Bayesian A/B testing, bandit algorithms for optimization, and more, which won't be covered in detail in this book.

Test statistics

There are many different test statistics that can be used in hypothesis testing. These include the Z-statistic, T-statistic, F-statistic, and chi-square statistic. You can, of course, implement these measures from scratch in Go without too much trouble. However, there are also some pre-existing implementations that you can use.

Returning to `gonum.org/v1/gonum/stat`, we can calculate a chi-square statistic as follows:

```
// Define observed and expected values. Most
// of the time these will come from your
// data (website visits, etc.).
observed := []float64{48, 52}
expected := []float64{50, 50}

// Calculate the ChiSquare test statistic.
chiSquare := stat.ChiSquare(observed, expected)
```

Calculating p-values

Let's say that we have the following scenario:

> *A survey of local residents reveals that 60% of all residents get no regular exercise, 25% exercise sporadically, and 15% exercise regularly. After doing some fancy modeling and putting some community services in place, the survey was repeated with the same questions. The follow up survey was completed by 500 residents with the following results:*
>
> *No Regular Exercise: 260*
> *Sporadic Exercise: 135*
> *Regular Exercise: 105*
> *Total: 500*

Now, we want to determine if there is evidence for a statistically significant shift in the responses of the residents. Our null and alternate hypotheses are as follows:

- H_0: The deviations from the previously observed percentages are due to pure chance
- H_a: The deviations are due to some underlying effect outside of pure chance (possibly our new community services)

First, let's calculate our test statistic using the chi-square test statistic:

```
// Define the observed frequencies.
observed := []float64{
    260.0, // This number is the number of observed with no regular
exercise.
    135.0, // This number is the number of observed with sporatic exercise.
    105.0, // This number is the number of observed with regular exercise.
}
// Define the total observed.
totalObserved := 500.0
// Calculate the expected frequencies (again assuming the null Hypothesis).
expected := []float64{
    totalObserved * 0.60,
    totalObserved * 0.25,
    totalObserved * 0.15,
}

// Calculate the ChiSquare test statistic.
chiSquare := stat.ChiSquare(observed, expected)

// Output the test statistic to standard out.
fmt.Printf("\nChi-square: %0.2f\n", chiSquare)
```

This will give us the following value for chi-square:

```
$ go build
$ ./myprogram

Chi-square: 18.13
```

Next, we need to calculate the p-value corresponding to this `Chi-square`. This requires us to know information about the **chi-squared distribution**, which defines the p-values for certain measures of chi-square and certain **degrees of freedom**. `github.com/gonum/stat` also includes a representation of this chi-squared distribution from which we can calculate our `p-value`:

```
// Create a Chi-squared distribution with K degrees of freedom.
// In this case we have K=3-1=2, because the degrees of freedom
// for a Chi-squared distribution is the number of possible
// categories minus one.
chiDist := distuv.ChiSquared{
    K: 2.0,
    Src: nil,
}

// Calculate the p-value for our specific test statistic.
pValue := chiDist.Prob(chiSquare)

// Output the p-value to standard out.
fmt.Printf("p-value: %0.4f\n\n", pValue)
```

This gives us the following result:

```
$ go build
$ ./myprogram

Chi-square: 18.13
p-value: 0.0001
```

So, there is a 0.01% probability that the results we see in the deviations in the second version of the survey are purely due to chance. If we, for example, are using a threshold of 5% (which is common), we would need to reject the null hypothesis and adopt our alternative hypothesis.

References

Vectors and matrices:

- `gonum.org/v1/gonum/floats` docs: `https://godoc.org/gonum.org/v1/gonum/floats`
- `gonum.org/v1/gonum/mat` docs: `https://godoc.org/gonum.org/v1/gonum/mat`

Statistics:

- `gonum.org/v1/gonum/stat` docs: `https://godoc.org/gonum.org/v1/gonum/stat`
- `github.com/montanaflynn/stats` docs: `https://godoc.org/github.com/montanaflynn/stats`

Visualization:

- `gonum.org/v1/plot` docs: `https://godoc.org/gonum.org/v1/plot`
- `gonum.org/v1/plot` wiki with examples: `https://github.com/gonum/plot/wiki/Example-plots`

Probability:

- `gonum.org/v1/gonum/stat/distuv` docs: `https://godoc.org/gonum.org/v1/gonum/stat/distuv`

Summary

This introduction to matrices, linear algebra, statistics, and probability in Go has given us a set of tools to understand, structure, and operate on data. This set of tools will be used throughout the book as we work on a diverse set of problems, and these tools could be used in a variety of contexts outside of machine learning. However, in the next chapter, we will discuss some ideas and techniques that will be extremely important in the machine learning context, specifically, evaluation and validation.

3
Evaluation and Validation

In order to have sustainable, responsible machine learning workflows and develop machine learning applications that produce true value, we need to be able to measure how well our machine learning models perform. We also need to ensure that our machine learning models generalize to data that they will see in production. If we don't do these things, we are basically shooting in the dark. We will have no understanding of the expected behavior of our models and we won't be able to improve them over time.

The process of measuring how a model is performing (with respect to certain data) is called **evaluation**. The process of ensuring that our model generalizes to data that we might expect to encounter is called **validation**. Both processes need to be present in every machine learning workflow and application, and we will cover both in this chapter.

Evaluation

A basic tenet of science is measurement, and the science of machine learning is not an exception. We need to be able to measure, or evaluate, how well our models are performing, so we can continue to improve on them, compare one model to another, and detect when our models are behaving poorly.

There's only one problem. How do we evaluate how our models are performing? Should we measure how fast they can be trained or make inferences? Should we measure how many times they get the right answer? How do we know what the right answer is? Should we measure how far we deviated from the observed values? How do we measure that distance?

As you can see, there are a lot of decisions to make around how we evaluate our models. What really matters is the context. In some cases, efficiency definitely matters, but every machine learning context requires us to measure how our predictions, inferences, or results match the ideal predictions, inferences, or results. Thus, measuring this comparison between computed results and ideal results should always take priority over speed optimizations.

Generally, there are some types of results that we will need to evaluate:

- **Continuous**: Results such as total sales, stock price, and temperature that can take any continuous numerical value ($12102.21, 92 degrees, and so on)
- **Categorical**: Results such as fraud/not fraud, activity, and name that can take one of a finite number of categories (fraud, standing, Frank, and so on)



Each of these types of results have corresponding evaluation metrics that will be covered here. However, remember that your choice of evaluation metric depends on what you are trying to achieve with your machine learning model. There is no one-size-fits-all metric, and in some cases, you may even need to create your own metric.

Continuous metrics

Let's say that we have a model that is supposed to predict some continuous value, like a stock price. Suppose that we have accumulated some predicted values that we can compare to actual observed values:

```
observation,prediction
22.1,17.9
10.4,9.1
9.3,7.8
18.5,14.2
12.9,15.6
7.2,7.4
11.8,9.7
...
```

Now, how do we measure the performance of this model? Well, the first step would be taking the difference between the observed and predicted values to get an `error`:

```
observation,prediction,error
22.1,17.9,4.2
10.4,9.1,1.3
9.3,7.8,1.5
18.5,14.2,4.3
12.9,15.6,-2.7
7.2,7.4,-0.2
11.8,9.7,2.1
...
```

The error gives us a general idea of *how far off we were* from the value that we were supposed to predict. However, it's not really feasible or practical to look at all the error values individually, especially when there is a lot of data. There could be a million or more of these error values. Thus, we need a way to understand the errors in aggregate.

The **mean squared error** (**MSE**) and **mean absolute error** (**MAE**) provide us with a view on errors in aggregate:

- MSE or **mean squared deviation** (**MSD**) is the average of the squares of all the errors
- MAE is the average of the absolute values of all the errors

Both MSE and MAE give us a good overall picture of how good our predictions are, but they do have differences. As the MSE takes the squares of the errors, large error values (for example, corresponding to outliers) are emphasized more than in the MAE. In other words, MSE is more sensitive to outliers. MAE, on the other hand, maintains the same units as the variable that we are trying to predict, and is thus directly comparable to these values.

For this dataset, we can parse the observed and predicted values and calculate the MAE and MSE as follows:

```
// Open the continuous observations and predictions.
f, err := os.Open("continuous_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values
```

```
// form the continuous data file.
var observed []float64
var predicted []float64

// line will track row numbers for logging.
line := 1
// Read in the records looking for unexpected types in the columns.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Skip the header.
    if line == 1 {
        line++
        continue
    }

    // Read in the observed and predicted values.
    observedVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    predictedVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    // Append the record to our slice, if it has the expected type.
    observed = append(observed, observedVal)
    predicted = append(predicted, predictedVal)
    line++
}

// Calculate the mean absolute error and mean squared error.
var mAE float64
var mSE float64
for idx, oVal := range observed {
    mAE += math.Abs(oVal-predicted[idx]) / float64(len(observed))
    mSE += math.Pow(oVal-predicted[idx], 2) / float64(len(observed))
}
```

```
// Output the MAE and MSE value to standard out.
fmt.Printf("\nMAE = %0.2f\n", mAE)
fmt.Printf("\nMSE = %0.2f\n\n", mSE)
```

For our example data, this results in the following:

```
$ go build
$ ./myprogram

MAE = 2.55

MSE = 10.51
```

To judge if these are good values or not, we need to compare them to the values in our observed data. In particular, the MAE is `2.55` and the mean of our observed values is 14.0, so our MAE is about 20% of our mean value. Not very good, depending on the context.

Along with the MSE and MAE, you will likely see **R-squared** (also known as $\mathbf{R^2}$ or **R2**), or the **coefficient of determination**, used as an evaluation metric for continuous variable models. R-squared also gives us a general idea about the deviations of our predictions, but the idea of R-squared is slightly different.

R-squared measures the proportion of the variance in the observed values that we capture in the predicted values. Remember that the values that we are trying to predict have some variability. For example, we might be trying to predict stock prices, interest rates, or disease progressions, which, by their very nature, aren't all the same. We are attempting to create a model that can predict this variability in the observed values, and the percentage of the variation that we capture is represented by R-squared.

Conveniently, `gonum.org/v1/gonum/stat` has a built-in function to calculate R-squared:

```
// Calculate the R^2 value.
rSquared := stat.RSquaredFrom(observed, predicted, nil)

// Output the R^2 value to standard out.
fmt.Printf("\nR^2 = %0.2f\n\n", rSquared)
```

Running the preceding code for our example dataset results in the following:

```
$ go build
$ ./myprogram

R^2 = 0.37
```

So, is this a good or bad R-squared? Remember that R-squared is a percentage and higher percentages are better. Here, we are capturing about 37% of the variance in the variable that we are trying to predict. Not very good.

Categorical metrics

Let's say that we have a model that is supposed to predict some discrete value, such as fraud/not fraud, standing/sitting/walking, approved/not approved, and so on. Our data might look something like the following:

```
observed,predicted
0,0
0,1
2,2
1,1
1,1
0,0
2,0
0,0
...
```

The observed values could take any one of a finite number of values (in this case 1, 2, or 3). Each of these values represents one of the discrete categories in our data (class 1 might correspond to a fraudulent transaction, class 2 might correspond to a transaction that is not fraudulent, and class 3 might correspond to an invalid transaction, for example). The predicted values could also take one of these discrete values. In evaluating our predictions, we want to somehow measure how right we were in making those discrete predictions.

Individual evaluation metrics for categorical variables

Actually, there are a huge number of ways to evaluate discrete predictions with metrics, including accuracy, precision, recall, specificity, sensitivity, fallout, false omission rate, and many more. As with continuous variables, there is no one-size-fits-all metric for evaluation. Each time you approach a problem, you need to determine the metric that fits the problem and matches the goals of the project. You don't want to optimize for the wrong things and then waste a bunch of time reimplementing your model based on other metrics.

To understand these metrics and determine which is appropriate for our use case, we need to realize that there are a number of different scenarios that could occur when we are making discrete predictions:

- **True Positive** (**TP**): We predicted a certain category, and the observation was actually that category (for example, we predicted fraud and the observation was fraud)
- **False Positive** (**FP**): We predicted a certain category, but the observation was actually another category (for example, we predicted fraud but the observation was not fraud)
- **True Negative** (**TN**): We predicted that the observation wasn't a certain category, and the observation was not that category (for example, we predicted not fraud and the observation was not fraud)
- **False Negative** (**FN**): We predicted that the observation wasn't a certain category, but the observation was actually that category (for example, we predicted not fraud but the observation was fraud)

You can see that there are a number of ways we can combine, aggregate, and measure these scenarios. In fact, we could even aggregate/measure them in some sort of unique way related to our specific problem. However, there are some pretty standard ways of aggregating and measuring these scenarios that result in the following common metrics:

- **Accuracy**: The percentage of predictions that were right, or *(TP + TN)/(TP + TN + FP + FN)*
- **Precision**: The percentage of positive predictions that were actually positive, or *TP/(TP + FP)*
- **Recall**: The percentage of positive predictions that were identified as positive, or *TP/(TP + FN)*



Even though I'm going to emphasize these here, you should take a look at other common metrics and their implications. A good overview can be found at `https://en.wikipedia.org/wiki/Precision_and_recall`.

The following is an example that parses our data and calculates accuracy. First, we read in our `labeled.csv` file, create a CSV reader, and initialize two slices that will hold our parsed observed/predicted values:

```
// Open the binary observations and predictions.
f, err := os.Open("labeled.csv")
if err != nil {
    log.Fatal(err)
```

```
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values
// form the labeled data file.
var observed []int
var predicted []int
```

Then we will iterate over the records in the CSV parsing the values, and we will compare the observed and predicted values to calculate accuracy:

```
// line will track row numbers for logging.
line := 1

// Read in the records looking for unexpected types in the columns.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Skip the header.
    if line == 1 {
        line++
        continue
    }

    // Read in the observed and predicted values.
    observedVal, err := strconv.Atoi(record[0])
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    predictedVal, err := strconv.Atoi(record[1])
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    // Append the record to our slice, if it has the expected type.
    observed = append(observed, observedVal)
```

```
        predicted = append(predicted, predictedVal)
        line++
    }

    // This variable will hold our count of true positive and
    // true negative values.
    var truePosNeg int

    // Accumulate the true positive/negative count.
    for idx, oVal := range observed {
        if oVal == predicted[idx] {
            truePosNeg++
        }
    }

    // Calculate the accuracy (subset accuracy).
    accuracy := float64(truePosNeg) / float64(len(observed))

    // Output the Accuracy value to standard out.
    fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

Running this results in the following:

```
$ go build
$ ./myprogram

Accuracy = 0.97
```

97%! That's pretty good. That means we were right 97% of the time.

We can similarly calculate precision and recall. However, you may have noticed that there are a couple of ways we can do this when we have more than two categories or classes. We could consider class 1 as positive and the other classes as negative, class 2 as positive and the other classes as negative, and so on. That is, we could calculate a precision or recall for each of our classes, as shown in the following code sample:

```
    // classes contains the three possible classes in the labeled data.
    classes := []int{0, 1, 2}

    // Loop over each class.
    for _, class := range classes {

        // These variables will hold our count of true positives and
        // our count of false positives.
        var truePos int
        var falsePos int
        var falseNeg int
```

```
    // Accumulate the true positive and false positive counts.
    for idx, oVal := range observed {

        switch oVal {

        // If the observed value is the relevant class, we should check to
        // see if we predicted that class.
        case class:
            if predicted[idx] == class {
                truePos++
                continue
            }

            falseNeg++

        // If the observed value is a different class, we should
        // check to see if we predicted a false positive.
        default:
            if predicted[idx] == class {
                falsePos++
            }
        }
    }

    // Calculate the precision.
    precision := float64(truePos) / float64(truePos+falsePos)

    // Calculate the recall.
    recall := float64(truePos) / float64(truePos+falseNeg)

    // Output the precision value to standard out.
    fmt.Printf("\nPrecision (class %d) = %0.2f", class, precision)
    fmt.Printf("\nRecall (class %d) = %0.2f\n\n", class, recall)
}
```

Running this code results in the following:

```
$ go build
$ ./myprogram

Precision (class 0) = 1.00
Recall (class 0) = 1.00


Precision (class 1) = 0.96
Recall (class 1) = 0.94
```

```
Precision (class 2) = 0.94
Recall (class 2) = 0.96
```

Notice that the precision and recall are slightly different metrics and have different implications. If we wanted to get an overall precision or recall, we could average the per-class precisions and recalls. In fact, if certain classes were more important than other classes, we could take a weighted average of these and use that as our evaluation metric.

You can see that a couple of the metrics are 100%. This seems good, but it might actually indicate a problem, as we will further discuss.



In some cases, such as finance and banking, false positives or other cases may be very costly for certain classes. For example, a mislabeling a transaction as fraudulent might result in significant losses. On the other hand, certain results for other classes might be negligible. These scenario might warrant the use of a custom metric or cost function that weights certain classes, certain results, or certain combinations of results as more important than others.

Confusion matrices, AUC, and ROC

In addition to calculating individual numerical metrics for our models, there are a variety of techniques to combine various metrics into a form that gives you a more complete representation of model performance. These include, but are certainly not limited to, **confusion matrices** and **area under the curve** (**AUC**)/**Receiver Operating Characteristic** (**ROC**) **curves**.

Confusion matrices allow us to visualize the various **TP**, **TN**, **FP**, and **FN** values that we predict in a two-dimensional format. A confusion matrix has rows corresponding to the categories that you were supposed to predict, and columns corresponding to categories that were predicted. Then, the value of each element is the corresponding count:

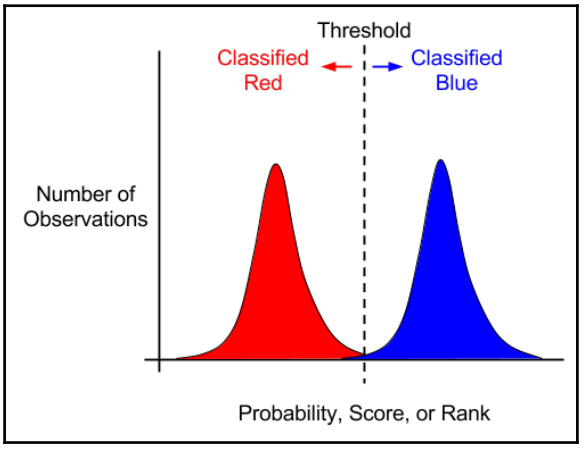
		Predicted	
		Fraud	Not Fraud
Observed	Fraud	TP	FN
	Not Fraud	FP	TN

As you can see, the ideal situation is that your confusion matrix only has entries on the diagonal (**TP**, **TN**). The diagonal elements represent predicting a certain category and the observation actually being in that category. The off-diagonal elements include counts for predictions that were incorrect.

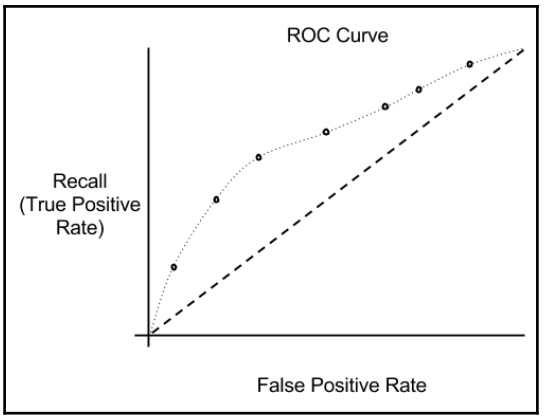
This type of confusion matrix can be especially useful for problems that have more than two categories. For example, you may be trying to predict various activities based on mobile accelerator and position data. These activities may include more than two categories, such as standing, sitting, running, driving, and so on. The confusion matrix for this problem with be larger than 2 x 2 and would allow you to quickly gauge the overall performance of your model on all categories, and identify categories in which your model is performing poorly.

In addition to confusion matrices, ROC curves are commonly used to get an overall picture of the performance of binary classifiers (or models that are trained to predict one of two categories). ROC curves plot the recall versus false positive rate (*FP/(FP + TN)*) for every possible classification threshold.

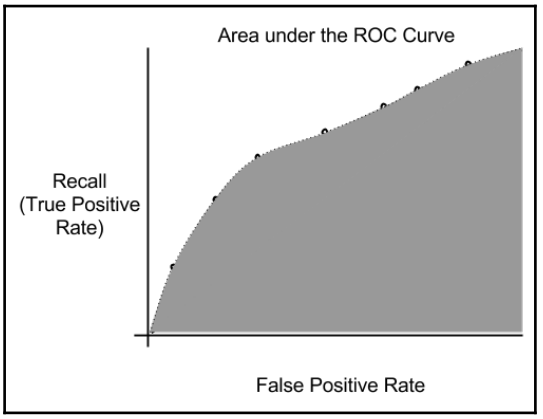
The thresholds used in an ROC curve represent various boundaries or rankings in which you are separating the two categories of your classification. That is, the model that is evaluated by the ROC curve must make a prediction for the two classes based on probability, ranking, or score (referred to as a score in the following image). In every example mentioned earlier, a score is classified one way, and vise versa:



To generate an ROC curve, we plot a point for each score or rank in our testing examples (recall, false positive rate). We can then connect these to form a curve. In many cases, you will see a straight line plotted down the diagonal of the ROC curve plot. This straight line is a reference line for a classifier, with approximately random predictive power:



A good ROC curve is one that is in the upper left section of the plot, which means that our model has better than random predictive power. The more that the ROC curve hugs the upper left hand side of the plot, the better. This means that good ROC curves have more AUC; AUC for ROC curves is also used as an evaluation metric. Refer to the following figure:



`gonum.org/v1/gonum/stat` has some built-in functions and types that help you build ROC curves and AUC metrics:

```
func ROC(n int, y []float64, classes []bool, weights []float64) (tpr, fpr
[]float64)
```

Here is a quick example that calculates the AUC for an ROC curve with gonum:

```
// Define our scores and classes.
scores := []float64{0.1, 0.35, 0.4, 0.8}
classes := []bool{true, false, true, false}

// Calculate the true positive rates (recalls) and
// false positive rates.
tpr, fpr := stat.ROC(0, scores, classes, nil)
// Compute the Area Under Curve.
auc := integrate.Trapezoidal(fpr, tpr)

// Output the results to standard out.
fmt.Printf("true positive rate: %v\n", tpr)
fmt.Printf("false positive rate: %v\n", fpr)
fmt.Printf("auc: %v\n", auc)
```

Running this code results in the following:

```
$ go build
$ ./myprogram
true positive rate: [0 0.5 0.5 1 1]
false positive rate: [0 0 0.5 0.5 1]
auc: 0.75
```

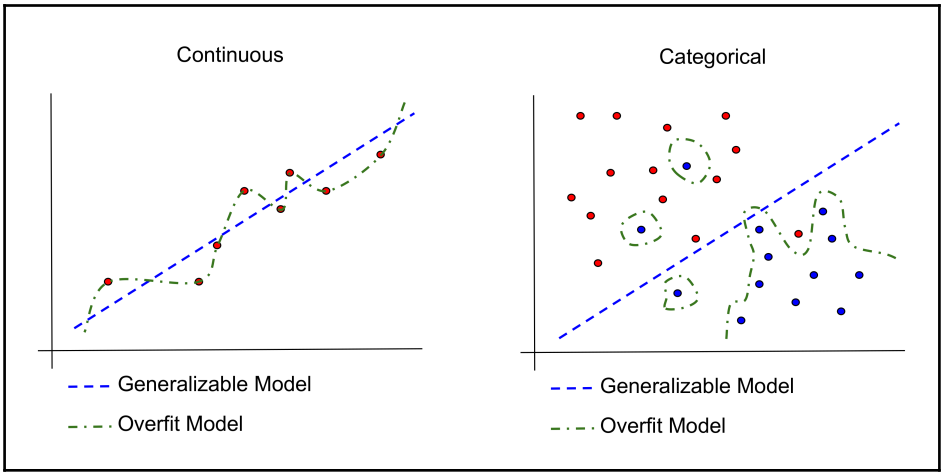
Validation

So, now we know some ways to measure how well our model is performing. In fact, if we wanted to, we could create a super sophisticated, complicated model that could predict every observation without error. For example, we could create a model that would take the index of the row of the observation and return the exact answer for each of those rows. It might be a really big function with a lot of parameters, but it would return the correct answers.

So, what's the problem with this? Well, the problem is that it would not generalize to new data. Our complicated model would predict really well for the data that we would expose it to, but once we try some new input data (that isn't part of our training dataset), the model would likely perform poorly.

We call this type of model (that doesn't generalize) a model that has been **overfit**. That is, our process of making the model more and more complicated based on the data that was available to us was overfitting the model.

Overfitting can happen when predicting continuous values or discrete/categorical values:



To prevent overfitting, we need to validate our model. There are multiple ways to perform validation, and we will cover a couple of these here.

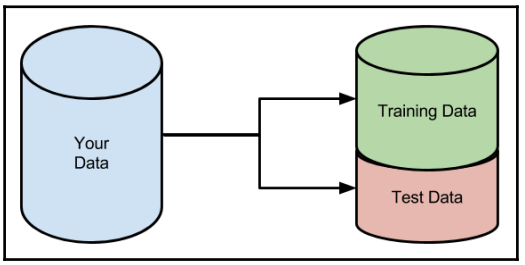


Every time you are productionizing a model, you need to ensure that you have validated your model and understand how it will generalize to new data.

Training and test sets

The first method to help prevent overfitting is to train or fit your model on a portion of your dataset and then test or evaluate your model on a different portion of your dataset. Training your model generally consists of parameterizing one or more functions that make up your model, such that the functions that predict what you are trying to predict. Then, you can evaluate this trained model using one or more of the evaluation metrics that we discussed previously. The important thing here is that you do not want to test/evaluate your model on the same data that is used to train your model.

By reserving part of your data for testing, you are simulating the scenario in which your model sees new data. That is, the model is making predictions based on data that was not used in parameterizing the model.

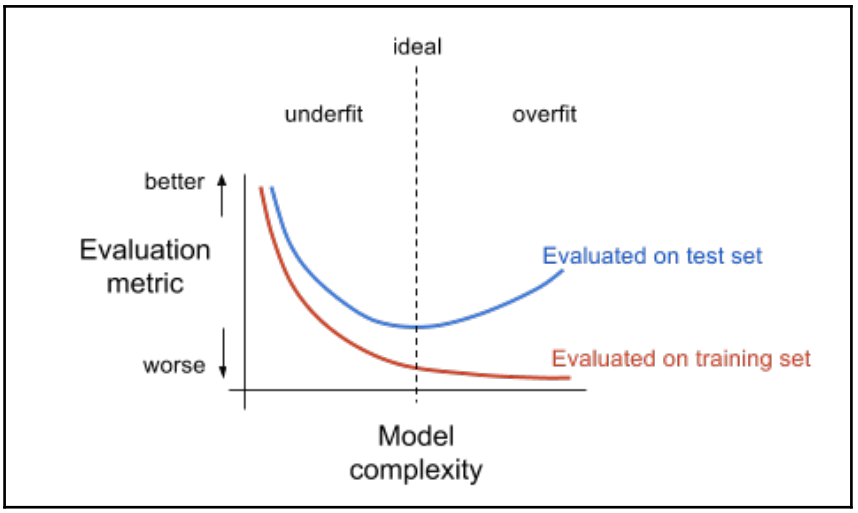


Many people start by splitting 80% of their data into a training data set and 20% into a test set (an 80/20 split). However, you will see different people splitting their datasets up in different proportions. The proportion of test to training data depends a little bit on the type and amount of data that you have and the model that you are trying to train. Generally, you want to ensure that both your training data and test data are a fairly accurate representation of your data on a large scale.

For example, if you are trying to predict one of a few different categories, A, B, and C, you wouldn't want your training data to include observations that only correspond to A and B. A model trained on such a dataset would likely only be able to predict the A and B categories. Likewise, you wouldn't want your test set to include some subset of the categories, or artificially weighted proportions of the categories. This could very easily happen, depending on how your data was generated.

In addition, you want to make sure that you have enough training data to reduce the variability in your determined parameters as they are computed over and over. If you have too few training data points, or poorly sampled training data points, your model training may produce parameters with a lot of variability, or it may not even be able to converge numerically. These are indications that your model lacks predictive power.

Typically, as you increase the complexity of your model, you will be able to improve the evaluation metric that you are using for your training data, but at some point, the evaluation metric will start getting worse for your test data. When the evaluation metric starts to get worse for your test data, you are starting to overfit your model. The ideal scenario is when you are able to increase your model complexity up to the inflection point, where the test evaluation metric starts to degrade. Another way of putting this (which fits very well into our general philosophy for model building in this book) is that we want the most interpretable model (or simplistic model) that can produce valuable results.



One way to quickly split a dataset into training and test sets is with `github.com/kniren/gota/dataframe`. Let's demonstrate this using a dataset, which includes a bunch of anonymized information about medical patients and a corresponding indication of the progression of disease and diabetes:

```
age,sex,bmi,map,tc,ldl,hdl,tch,ltg,glu,y
0.0380759064334,0.0506801187398,0.0616962065187,0.021872354995,-0.044223498
4244,-0.0348207628377,-0.043400845652,-0.00259226199818,0.0199084208763,-0.
0176461251598,151.0
-0.00188201652779,-0.044641636507,-0.0514740612388,-0.0263278347174,-0.0084
4872411122,-0.0191633397482,0.0744115640788,-0.0394933828741,-0.06832974362
44,-0.0922040496268,75.0
0.0852989062967,0.0506801187398,0.0444512133366,-0.00567061055493,-0.045599
4512826,-0.0341944659141,-0.0323559322398,-0.00259226199818,0.0028637705189
4,-0.0259303389895,141.0
-0.0890629393523,-0.044641636507,-0.0115950145052,-0.0366564467986,0.012190
5687618,0.0249905933641,-0.0360375700439,0.0343088588777,0.0226920225667,-0
.00936191133014,206.0
0.00538306037425,-0.044641636507,-0.0363846922045,0.021872354995,0.00393485
161259,0.0155961395104,0.00814208360519,-0.00259226199818,-0.0319914449414,
-0.0466408735636,135.0
...
```

You can retrieve this data set here: `https://archive.ics.uci.edu/ml/datasets/diabetes`.

To split this data with `github.com/kniren/gota/dataframe`, we can do the following (where we save the training and test splits to respective CSV files):

```
// Open the diabetes dataset file.
f, err := os.Open("diabetes.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
diabetesDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * diabetesDF.Nrow()) / 5
testNum := diabetesDF.Nrow() / 5
if trainingNum+testNum < diabetesDF.Nrow() {
    trainingNum++
}

// Create the subset indices.
trainingIdx := make([]int, trainingNum)
testIdx := make([]int, testNum)

// Enumerate the training indices.
for i := 0; i < trainingNum; i++ {
    trainingIdx[i] = i
}

// Enumerate the test indices.
for i := 0; i < testNum; i++ {
    testIdx[i] = trainingNum + i
}

// Create the subset dataframes.
trainingDF := diabetesDF.Subset(trainingIdx)
testDF := diabetesDF.Subset(testIdx)

// Create a map that will be used in writing the data
// to files.
setMap := map[int]dataframe.DataFrame{
    0: trainingDF,
    1: testDF,
}

// Create the respective files.
for idx, setName := range []string{"training.csv", "test.csv"} {
```

```
        // Save the filtered dataset file.
        f, err := os.Create(setName)
        if err != nil {
            log.Fatal(err)
        }

        // Create a buffered writer.
        w := bufio.NewWriter(f)

        // Write the dataframe out as a CSV.
        if err := setMap[idx].WriteCSV(w); err != nil {
            log.Fatal(err)
        }
    }
```

Running this results in the following:

```
$ go build
$ ./myprogram
$ wc -l *.csv
   443 diabetes.csv
    89 test.csv
   355 training.csv
   887 total
```

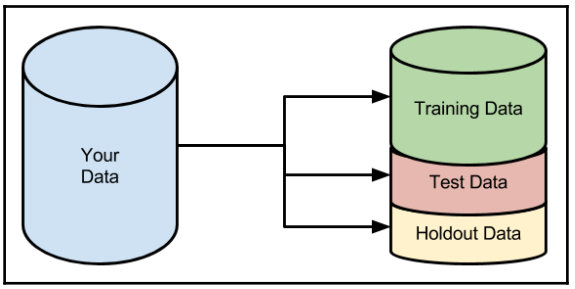
Holdout set

We are making progress to ensure that our models generalize using training and test sets. However, imagine the following scenario:

1. We develop a first version of our model based on our training set.
2. We test this first version of our model on the test set.
3. We aren't satisfied with the result on the test set, so we loop back to step 1 and repeat.

This process might seem logical, but you are probably already seeing a problem that can result from this procedure. We can actually overfit our model on the test data by iteratively exposing the model to our test set.

There are a couple of ways to deal with this extra level of overfitting. The first is by simply creating another split of our data called a **holdout set** (also known as a **validation set**). So, now we would have a training set, test set, and holdout set. This is sometimes called the three dataset validation, for obvious reasons.



Keep in mind that, to truly get an idea about the general performance of your model, your holdout set must never be used in training and testing. You should reserve this dataset for validation after you have gone through the process of training your model, making adjustments to the model, and getting an acceptable performance on the test dataset.

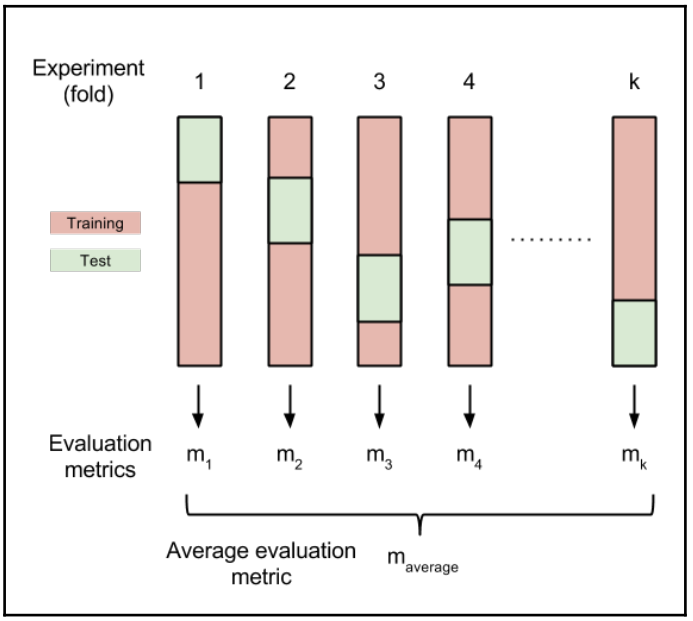


You might be wondering how you can manage this splitting of data over time and recover different sets of data used to train or test certain models. This "provenance" of data is crucial when trying to maintain integrity in your machine learning workflows. This is also exactly what Pachyderm's data versioning (introduced in Chapter 1, *Gathering and Organizing Data*) was created to handle. We will see exactly how this plays out at scale later in Chapter 9, *Deploying and distributing Analyses and Models*.

Cross validation

In addition to reserving a holdout set for validation, cross validation is a common technique to validate the generality of a model. In cross validation, or k-fold cross validation, you actually perform *k* random splits of your dataset into different training and test combinations. Think of these as *k* experiments.

Once you have performed each split, you train your model on the training data for that split, and then evaluate it on the test data for that split. This process results in an evaluation metric result for each random split of your data. You can then average these evaluation metrics to get an overall evaluation metric that is a more general representation of model performance than any one of the individual evaluation metrics by themselves. You can also look at the variance in the evaluation metrics to get an idea about the stability of your various experiments. This process is illustrated in the following image:



Some advantages of using cross validation, in comparison to dataset validation, are as follows:

- You are making use of your entire dataset, and thus, are actually exposing your model to more training examples and more testing examples
- There are some convenience functions and packaging already written for cross validation
- It helps prevent the biases that may result from choosing a single validation set

The `github.com/sjwhitworth/golearn` is one Go package that provides some convenience functions for cross validation. Actually, `github.com/sjwhitworth/golearn` includes a bunch of machine learning functionality that we will cover later on in the book, but for now, let's just look at what functionality is available for cross validation.

If you look at the `github.com/sjwhitworth/golearn/evaluation` package Godocs, you will see the following function that is available for cross validation:

```
func GenerateCrossFoldValidationConfusionMatrices(data base.FixedDataGrid,
cls base.Classifier, folds int) ([]ConfusionMatrix, error)
```

This function can actually be used with a variety of models, but here is an example using a decision tree model (don't worry about the details of the model here):

```
// Define the decision tree model.
tree := trees.NewID3DecisionTree(param)

// Perform the cross validation.
cfs, err := evaluation.GenerateCrossFoldValidationConfusionMatrices(myData,
tree, 5)
if err != nil {
    panic(err)
}

// Calculate the metrics.
mean, variance := evaluation.GetCrossValidatedMetric(cfs,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the results to standard out.
fmt.Printf("%0.2f\t\t%.2f (+/- %.2f)\n", param, mean, stdev*2)
```

References

Evaluation:

- Essay on overfitting: `http://scott.fortmann-roe.com/docs/MeasuringError.html`
- Essay on bias-variance trade-off: `http://scott.fortmann-roe.com/docs/BiasVariance.html`
- Comparison of categorical evaluation metrics: `https://en.wikipedia.org/wiki/Precision_and_recall`
- `gonum.org/v1/gonum/stat` docs: `https://godoc.org/gonum.org/v1/gonum/stat`
- `github.com/sjwhitworth/golearn/evaluation` docs: `https://godoc.org/github.com/sjwhitworth/golearn/evaluation`

Validation:

- `github.com/kniren/gota/dataframe` docs: `https://godoc.org/github.com/kniren/gota/dataframe`
- `github.com/sjwhitworth/golearn/evaluation` docs: `https://godoc.org/github.com/sjwhitworth/golearn/evaluation`

Summary

Choosing an appropriate evaluation metric and laying out a procedure for evaluation/validation are essential parts of any machine learning project. You have learned about a variety of relevant evaluation metrics and how to avoid overfitting using holdout sets and/or cross validation. In the next chapter, we will start looking at machine learning models and we will build our first model using linear regression!

4
Regression

The first group of machine learning techniques that we will explore is generally referred to as **regression**. Regression is a process through which we can understand how one variable (for example, sales) changes with respect to another variable (for example, number of users). These techniques are useful on their own. However, they are also a good starting point to discuss machine learning techniques because they form the basis of other, more complicated, techniques that we will discuss later in the book.

Generally, regression techniques in machine learning are concerned with predicting continuous values (for example, stock price, temperature, or disease progression). **Classification**, which we will cover in the next chapter, is concerned with predicting discrete variables, or one of a discrete set of categories (for example, fraud/not fraud, sitting/standing/running, or hot dog/not hot dog). As mentioned, regression techniques are used throughout machine learning as part of classification algorithms, but in this chapter we will focus on their basic application to predict continuous values.

Understanding regression model jargon

As already mentioned, regression itself is a process to analyze a relationship between one variable and another variable, but there are some terms used in machine learning to describe these variables along with various types of regression and processes associated with regression:

- **Response** or **dependent variable**: These terms will be used interchangeably for the variable that we are trying to predict based on one or more other variables. This variable is often labeled y.

- **Explanatory variables**, **independent variables**, **features**, **attributes**, or **regressors**: These terms will be used interchangeably for the variables that we are using to predict the response. These variables are often labeled x or x_1, x_2, and so on.
- **Linear regression**: This type of regression assumes that the dependent variable depends on the independent variable linearly (that is, following the equation for a line).
- **Nonlinear regression**: This type of regression assumes that the dependent variable depends on the independent variable in a relationship that is not linear (for example, polynomial or exponential).
- **Multiple regression**: A regression with more than one independent variable.
- **Fitting** or **training**: The process of parameterizing a model, such as a regression model, so that it can predict a certain dependent variable.
- **Prediction**: The process of using a parameterized model, such as a regression model, to predict a certain dependent variable.

Some of these terms will be used both in the context of regression and in other contexts throughout the rest of the book.

Linear regression

Linear regression is one of the most simple machine learning models. However, you should not dismiss this model by any means. As mentioned previously, it is an essential building block that is utilized in other models, and it has some very important advantages.

As discussed throughout this book, integrity in machine learning applications is crucial, and the simpler and more interpretable a model is, the easier it is to maintain integrity. In addition, because the model is simple and interpretable, it allows you to understand inferred relationships between variables and check your work mentally as you develop. In the words of Mike Lee Williams from Fast Forward Labs (in `http://blog.fastforwardlabs.com/2017/08/02/interpretability.html`):

> *The future is algorithmic. Interpretable models offer a safer, more productive, and ultimately more collaborative relationship between humans and intelligent machines.*

Linear regression models are interpretable, and thus, they can provide a safe and productive option for data scientists. When you are searching for a model to predict a continuous variable, you should consider and try linear regression (or even multiple linear regression) if your data and problem allows you to use it.

Overview of linear regression

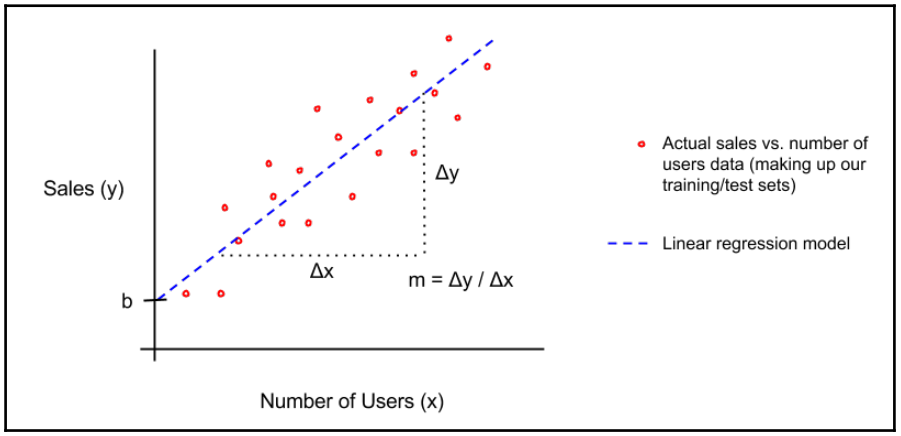
In linear regression, we attempt to model our dependent variable, *y*, by an independent variable, *x*, using the equation for a line:

$$y = mx + b$$

Here, *m* is the slope of the line and *b* is the intercept. For example, let's say that we want to model daily *sales* by the *number of users* on our website each day. To do this with linear regression, we would want to determine an *m* and *b* that would allow us to predict sales via the following formula:

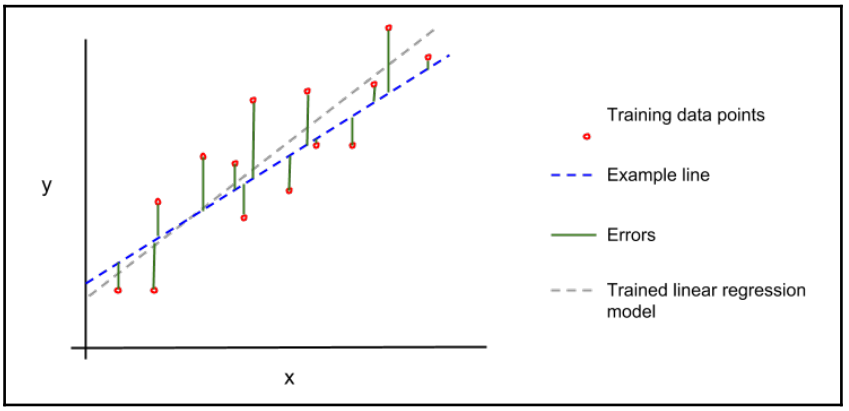
$$sales = m * (number\ of\ users) + b$$

Thus, our trained model is really just this parameterized function. We put in a **Number of Users** and we get the predicted **Sales**, as follows:



The training or fitting of a linear regression model involves determining the values of *m* and *b*, such that the resulting formula has predictive power for our response. There are a variety of methods to determine *m* and *b*, but the most common and simple method is called **ordinary least squares** (**OLS**).

To find *m* and *b* with OLS, we first pick a value for *m* and *b* to create a first example line. We then measure the vertical distance between each of our known points (for example, from our training set) and the example line. These distances are called **errors** or **residuals**, similar to the errors that we discussed in `Chapter 3`, *Evaluation and Validation*, and are illustrated in the following figure:



Next, we add up the sum of the squares of these errors:

$$\frac{error_1^2 + error_2^2 + \ldots + error_N^2}{N}$$

We adjust *m* and *b* until we minimize this sum of the squares of the errors. In other words, our trained linear regression line is the line that minimizes this sum of the squares.

There are a variety of methods to find the line that minimizes the sum of the squared errors and, for OLS, the line can be found analytically. However, a very popular and general optimization technique that is used to minimize the sum of the squared error is called **gradient descent**. This method can be more efficient in terms of implementation, advantageous computationally (in terms of memory, for example), and more flexible than analytic solutions.

Gradient descent is discussed in more detail in the `Appendix`, *Algorithms/Techniques Related to Machine Learning*, so we will avoid a lengthy discussion here. Suffice it to say that many implementations of linear and other regressions utilize gradient descent for the fitting or training of the linear regression line. In fact, gradient descent is ubiquitous in machine learning and also powers much more complicated modeling techniques such as deep learning.

Linear regression assumptions and pitfalls

Like all machine learning models, linear regression does not work in all situations and it does make certain assumptions about your data and the relationships in your data. The assumptions of linear regression are as follows:

- **Linear relationship**: This might seem obvious, but linear regression assumes that your dependent variable depends on your independent variable linearly (by means of the equation for a line). If this relationship is not linear, linear regression will likely perform poorly.
- **Normality**: This assumption means that your variables should be distributed according to a normal distribution (which looks like a bell shape). We will come back to this property later in the chapter and discuss some trade-offs and options when encountering non-normally distributed variables.
- **No multicollinearity**: Multicollinearity is a fancy term that means that independent variables are not really independent. They depend on each other in some fashion.
- **No auto-correlation**: Auto-correlation is another fancy term that means that a variable depends on itself or some shifted version of itself (like in some predictable time series).
- **Homoscedasticity**: This may be the fanciest word of this bunch of terms, but it means something relatively simple and is not really something you have to worry about very often. Linear regression assumes that the variance of your data is about the same around the regression line for all values of your independent variable.

Technically, all of these assumptions need to be fulfilled for us to use linear regression. It's very important that we know how our data is distributed and how it behaves. We will look into these assumptions when we profile data in an example use of linear regression.

As a data scientist or analyst, you want to keep the following pitfalls in mind as you apply linear regression:

- You are training your linear regression model for a certain range of your independent variable. You should be careful making predictions for values outside of this range because your regression line might not be applicable (for example, your dependent variable may start behaving non-linearly at extreme values).

- You can misspecify a linear regression model by finding some spurious relationship between two variables that really have nothing to do with one another. You should check to make sure that there is some logical reason why variables might be functionally related.
- Outliers or extreme values in your data may throw off a regression line for certain types of fitting, such as OLS. There are ways to fit a regression line that is more immune to outliers, or behaves differently with respect to outliers, such as orthogonal least squares or ridge regression.

Linear regression example

To illustrate linear regression, let's take an example problem and create our first machine learning model! The example data that we are going to use is example advertising data. It is in the `.csv` format and looks as follows:

```
$ head Advertising.csv
TV,Radio,Newspaper,Sales
230.1,37.8,69.2,22.1
44.5,39.3,45.1,10.4
17.2,45.9,69.3,9.3
151.5,41.3,58.5,18.5
180.8,10.8,58.4,12.9
8.7,48.9,75,7.2
57.5,32.8,23.5,11.8
120.2,19.6,11.6,13.2
8.6,2.1,1,4.8
```

The dataset includes a set of attributes representing spend on advertising outlets (`TV`, `Radio`, and `Newspaper`) along with corresponding sales (`Sales`). Our goal in this example will be to model the sales (our dependent variable) by one of the attributes of advertising spend (our independent variable).

Profiling the data

To make sure that we create a model, or at least process, that we understand, and to make sure that we can mentally check our results, we need to start every machine learning model building process with data profiling. We need to gain an understanding of how each of our variables are distributed and their range and variability.

To do this, we will calculate the summary statistics that we discussed earlier in `Chapter 2`, *Matrices, Probability, and Statistics*. Here, we will utilize a method built into the `github.com/kniren/gota/dataframe` package to calculate our summary statistics for all of the columns of our dataset in one operation:

```
// Open the CSV file.
advertFile, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer advertFile.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(advertFile)

// Use the Describe method to calculate summary statistics
// for all of the columns in one shot.
advertSummary := advertDF.Describe()

// Output the summary statistics to stdout.
fmt.Println(advertSummary)
```

Compiling and running this gives the following result:

```
$ go build
$ ./myprogram
[7x5] DataFrame

    column   TV         Radio     Newspaper  Sales
 0: mean     147.042500 23.264000 30.554000  14.022500
 1: stddev   85.854236  14.846809 21.778621  5.217457
 2: min      0.700000   0.000000  0.300000   1.600000
 3: 25%      73.400000  9.900000  12.600000  10.300000
 4: 50%      149.700000 22.500000 25.600000  12.900000
 5: 75%      218.500000 36.500000 45.100000  17.400000
 6: max      296.400000 49.600000 114.000000 27.000000
    <string> <float>    <float>   <float>    <float>
```

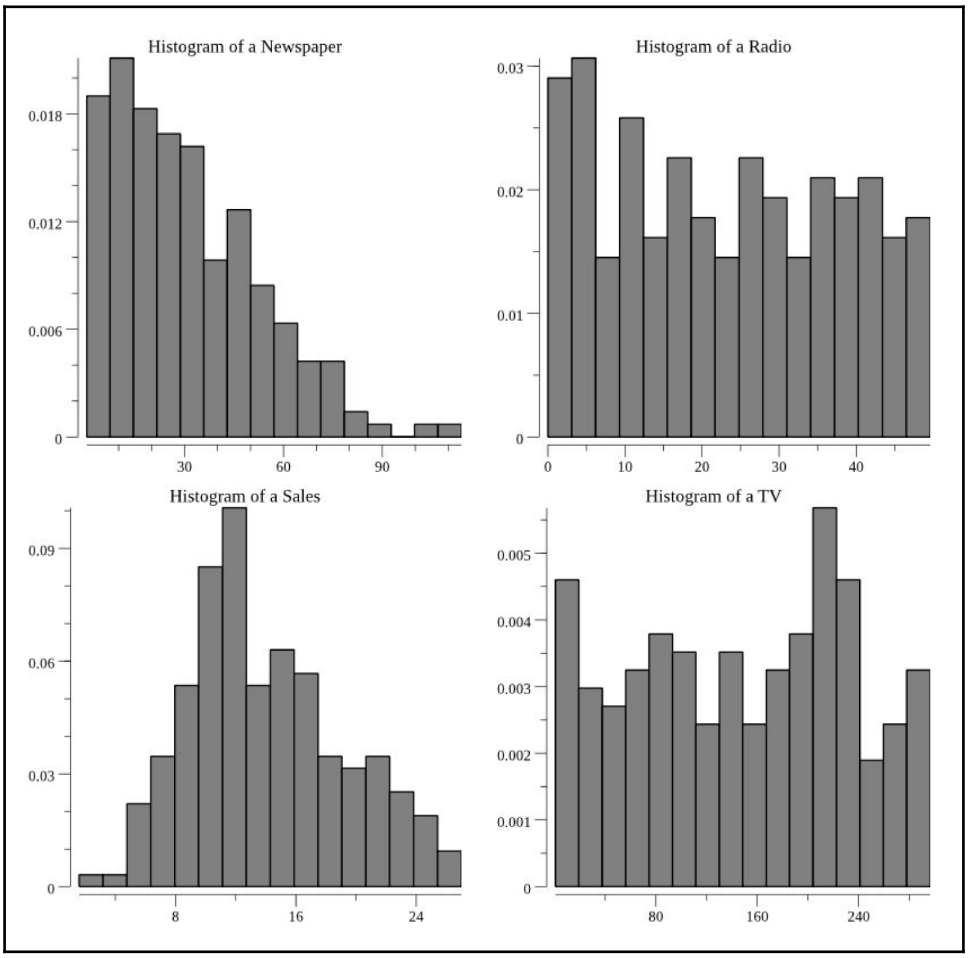
As you can see, this prints out all of our summary statistics in a nice tabular form and includes mean, standard deviation, minimum value, maximum value, *25%*/*75%* percentiles, and median (or 50% percentile).

These values give us a good numerical reference for the numbers that we will be seeing as we train our linear regression model. However, this does not give us a very good visual understanding of our data. For this, we will create a histogram for the values in each of the columns:

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(f)

// Create a histogram for each of the columns in the dataset.
for _, colName := range advertDF.Names() {

    // Create a plotter.Values value and fill it with the
    // values from the respective column of the dataframe.
    plotVals := make(plotter.Values, advertDF.Nrow())
    for i, floatVal := range advertDF.Col(colName).Float() {
        plotVals[i] = floatVal
    }

    // Make a plot and set its title.
    p, err := plot.New()
    if err != nil {
        log.Fatal(err)
    }
    p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

    // Create a histogram of our values drawn
    // from the standard normal.
    h, err := plotter.NewHist(plotVals, 16)
    if err != nil {
        log.Fatal(err)
    }

    // Normalize the histogram.
    h.Normalize(1)

    // Add the histogram to the plot.
    p.Add(h)

    // Save the plot to a PNG file.
    if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err != nil
{
```

```
            log.Fatal(err)
        }
    }
```

This program will create a `.png` image for each histogram:



Now, looking at these histograms and the summary statistics that we calculated, we need to consider if we are working within the assumptions of linear regression. In particular, we can see that not all of our variables are normally distributed (that is, they are in a bell shape). The sales might be somewhat bell-shaped, but the others do not look to be normal.

We could use a statistical tool, such as a **quantile-quantile** (**q-q**) plot, to determine how close the distributions are to normal distributions, and we could even perform a statistical test to determine the probability of the variables following a normal distribution. However, most of the time, we can get a general idea from the histograms.

Now we have to make a decision. At least some of our data does not technically fit within the assumptions of our linear regression model. We could now do one of the following:

- Try to transform our variables (with, for example, a power transformation) that follow a normal distribution, and then use these transformed variables in our linear regression model. The advantage of this option is that we would be operating within the assumptions of the model. The disadvantage is that we would be making our model harder to understand, and less interpretable.
- Get different data to solve our problem.
- Ignore our issue with the linear regression assumptions and try to create the model.

There may be other views on this, but my recommendation is that you try the third option first. There is not much harm in this option because you can train the linear regression model quickly. If you end up with a model that performs nicely, you have avoided further complications and have a nice simple model. If you end up with a model that performs poorly, you might need to resort to one of the other options.

Choosing our independent variable

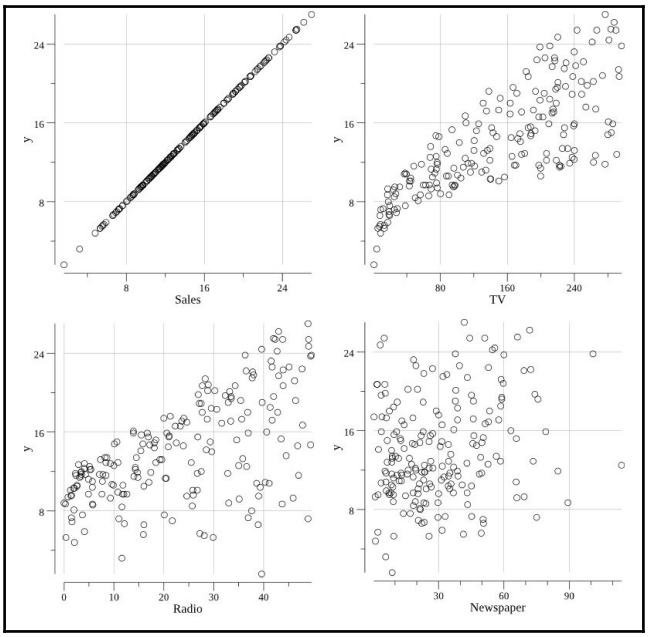
So, now we have some intuition about our data and have come to terms with how our data fits within the assumptions of the linear regression model. Now, how do we choose which variable to use as our independent variable in trying to predict our dependent variable, and average points per game?

The easiest way to make this decision is by visually exploring the correlation between the dependent variable and all of the choices that you have for independent variables. In particular, you can make scatter plots (using `gonum.org/v1/plot`) of your dependent variable versus each of the other variables:

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
```

```
advertDF := dataframe.ReadCSV(f)

// Extract the target column.
yVals := advertDF.Col("Sales").Float()

// Create a scatter plot for each of the features in the dataset.
for _, colName := range advertDF.Names() {
    // pts will hold the values for plotting
    pts := make(plotter.XYs, advertDF.Nrow())

    // Fill pts with data.
    for i, floatVal := range advertDF.Col(colName).Float() {
        pts[i].X = floatVal
        pts[i].Y = yVals[i]
    }

    // Create the plot.
    p, err := plot.New()
    if err != nil {
        log.Fatal(err)
    }
    p.X.Label.Text = colName
    p.Y.Label.Text = "y"
    p.Add(plotter.NewGrid())

    s, err := plotter.NewScatter(pts)
    if err != nil {
        log.Fatal(err)
    }
    s.GlyphStyle.Radius = vg.Points(3)

    // Save the plot to a PNG file.
    p.Add(s)
    if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_scatter.png"); err !=
nil {
        log.Fatal(err)
    }
}
```

This will create the following scatter plots:



As we look at these scatter plots, we want to deduce which of the attributes (**TV**, **Radio**, and/or **Newspaper**) have a linear relationship with our dependent variable, **Sales**. That is, could we draw a line on any of these scatter plots that would fit the trend of **Sales** versus the respective attribute? This is not always possible, and it likely will not be possible for all of the attributes that you have to work with for a given problem.

In this case, both **Radio** and **TV** appear to be somewhat linearly correlated with **Sales**. **Newspaper** may be slightly correlated with **Sales**, but the correlation is far from obvious. The linear relationship with **TV** seems most obvious, so let's start out with **TV** as our independent variable in our linear regression model. This would make our linear regression formula as follows:

$$Sales = m\ TV + b$$

One other thing to note here is that the variable **TV** might not be strictly homoscedastic, which was discussed earlier as an assumption of linear regression. This is worth noting (and likely worth documenting in the project), but we will continue on to see if we can create the linear regression model with some predictive power. We can always revisit this assumption if our model is behaving poorly, as a possible explanation.

Creating our training and test sets

To avoid overfitting and make sure that our model can generalize, we are going to split our dataset into a training set and a test set, as was discussed in `Chapter 3`, *Evaluation and Validation*. We will not bother with a holdout set here, because we are only going to make one pass through our model training without an iterative back and forth between training and testing. However, if you are experimenting with various dependent variables and/or iteratively adjusting any parameters of your model, you would want to create a holdout set that you save until the end of your model development process for validation.

We will use `github.com/kniren/gota/dataframe` to create our training and test datasets and then save them to respective `.csv` files. In this case, we will use an 80/20 split for our training and test data:

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
advertDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * advertDF.Nrow()) / 5
testNum := advertDF.Nrow() / 5
if trainingNum+testNum < advertDF.Nrow() {
    trainingNum++
}

// Create the subset indices.
trainingIdx := make([]int, trainingNum)
testIdx := make([]int, testNum)

// Enumerate the training indices.
for i := 0; i < trainingNum; i++ {
    trainingIdx[i] = i
}

// Enumerate the test indices.
for i := 0; i < testNum; i++ {
    testIdx[i] = trainingNum + i
}

// Create the subset dataframes.
```

```
trainingDF := advertDF.Subset(trainingIdx)
testDF := advertDF.Subset(testIdx)

// Create a map that will be used in writing the data
// to files.
setMap := map[int]dataframe.DataFrame{
    0: trainingDF,
    1: testDF,
}

// Create the respective files.
for idx, setName := range []string{"training.csv", "test.csv"} {

    // Save the filtered dataset file.
    f, err := os.Create(setName)
    if err != nil {
        log.Fatal(err)
    }

    // Create a buffered writer.
    w := bufio.NewWriter(f)

    // Write the dataframe out as a CSV.
    if err := setMap[idx].WriteCSV(w); err != nil {
        log.Fatal(err)
    }
}
```

This code will output the following training and test sets that we will use:

```
$ wc -l *.csv
  201 Advertising.csv
   41 test.csv
  161 training.csv
  403 total
```



The data that we were using here was not sorted or ordered by data in any fashion. However, if you are dealing with data that is sorted by response, by date, or in any other way, it is important that you randomly split your data into training and test sets. If you do not do this, your training and test sets may include only certain ranges of the response, may be influenced artificially by time/date, and so on.

Training our model

Next, we are going to actually train, or fit, our linear regression model. If you remember, this just means that we are finding the slope (*m*) and intercept (*b*) for the line that minimizes the sum of the squared errors. To perform this training, we will use a really great package from Sajari: `github.com/sajari/regression`. Sajari is a web search company that relies heavily on Go and machine learning, and they use `github.com/sajari/regression` in production.

To train a regression model using `github.com/sajari/regression`, we need to initialize a `regression.Regression` value, set a couple of labels, and fill the `regression.Regression` value with labeled training data points. After this, training our linear regression model is as easy as calling the `Run()` method on the `regression.Regression` value:

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
trainingData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// In this case we are going to try and model our Sales (y)
// by the TV feature plus an intercept. As such, let's create
// the struct needed to train a model using github.com/sajari/regression.
var r regression.Regression
r.SetObserved("Sales")
r.SetVar(0, "TV")

// Loop of records in the CSV, adding the training data to the regression
value.
for i, record := range trainingData {

    // Skip the header.
    if i == 0 {
        continue
```

```
        }

        // Parse the Sales regression measure, or "y".
        yVal, err := strconv.ParseFloat(record[3], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the TV value.
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Add these points to the regression value.
        r.Train(regression.DataPoint(yVal, []float64{tvVal}))
    }

    // Train/fit the regression model.
    r.Run()

    // Output the trained model parameters.
    fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)
```

Compiling and running this will result in the trained linear regression formula being printed to `stdout`:

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 7.07 + TV*0.05
```

Here, we can see that the package determined our linear regression line with an intercept of `7.07` and a slope of `0.5`. We can perform a little mental check here, because we saw in the scatter plots how the correlation between **TV** and **Sales** was up and to the right (that is, a positive correlation). This means that the slope should be positive in the formula, which it is.

Evaluating the trained model

We now need to measure the performance of our model to see if we really have any power to predict **Sales** using **TV** as in independent variable. To do this, we can load in our test set, make predictions using our trained model for each test example, and then calculate one of the evaluation metrics discussed in `Chapter 3`, *Evaluation and Validation*.

For this problem, let's use the Mean Absolute Error (MAE) as our evaluation metric. This seems reasonable, because it results in something directly comparable with our `Sales` values and we do not have to be too worried about outliers or extreme values.

To calculate the predicted **Sales** values using our trained `regression.Regression` value, we just need to parse the values in our test set and call the `Predict()` method on the `regression.Regression` value. We will then take the difference of these predicted values from the observed values, get the absolute value of the difference, and then add up all of the absolute values to get the MAE:

```
// Open the test dataset file.
f, err = os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV reader reading from the opened file.
reader = csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the test data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the observed Sales, or "y".
    yObserved, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
```

```
        }

        // Predict y with our trained model.
        yPredicted, err := r.Predict([]float64{tvVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
```

Compiling and running this evaluation gives the following result:

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 7.07 + TV*0.05

MAE = 3.01
```

How do we know if `MAE = 3.01` is good or bad? This is, again, why having a good mental model of your data is important. If you remember, we already computed the mean, range, and standard deviation of sales. The mean sales value was `14.02` and the standard deviation was `5.21`. Thus, our MAE is less than the standard deviations of our sales values and is about 20% of the mean value, and our model has some predictive power.

So, congratulations! We have built our first machine learning model that has predictive power!

To get better intuition about how our model is performing, we can also create a plot to help us visualize the linear regression line. This can be done with `gonum.org/v1/plot`. First, however, let's create a predict function that allows us to make our predictions without importing `github.com/sajari/regression`. This gives us a lightweight, in-memory version of the trained model:

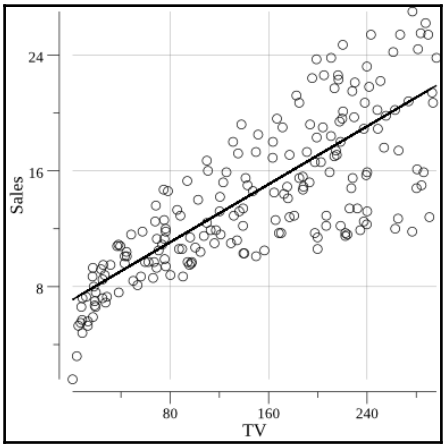
```
// predict uses our trained regression model to made a prediction.
func predict(tv float64) float64 {
    return 7.07 + tv*0.05
}
```

Then, we can create the visualization of our regression line:

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(f)

// Extract the target column.
yVals := advertDF.Col("Sales").Float()

// pts will hold the values for plotting.
pts := make(plotter.XYs, advertDF.Nrow())

// ptsPred will hold the predicted values for plotting.
ptsPred := make(plotter.XYs, advertDF.Nrow())

// Fill pts with data.
for i, floatVal := range advertDF.Col("TV").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
    ptsPred[i].X = floatVal
    ptsPred[i].Y = predict(floatVal)
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "TV"
p.Y.Label.Text = "Sales"
p.Add(plotter.NewGrid())

// Add the scatter plot points for the observations.
s, err := plotter.NewScatter(pts)
if err != nil {
    log.Fatal(err)
}
s.GlyphStyle.Radius = vg.Points(3)

// Add the line plot points for the predictions.
l, err := plotter.NewLine(ptsPred)
if err != nil {
```

```
        log.Fatal(err)
    }
    l.LineStyle.Width = vg.Points(1)
    l.LineStyle.Dashes = []vg.Length{vg.Points(5), vg.Points(5)}

    // Save the plot to a PNG file.
    p.Add(s, l)
    if err := p.Save(4*vg.Inch, 4*vg.Inch, "regression_line.png"); err != nil {
        log.Fatal(err)
    }
```

It will produce this plot when compiled and run:



As you can see, our trained linear regression line follows the linear trend of the real data points. This is another visual confirmation that we are on the right track!

Multiple linear regression

Linear regression is not limited to simple formulas of lines that depend on only one independent variable. Multiple linear regression is similar to what we discussed previously, but here we have multiple independent variables (x_1, x_2, and so on). In this case, our simple equation of a line is as follows:

$$y = m_1 x_1 + m_1 x_2 + \ldots + m_N x_N + b$$

Here, the x's are the various independent variables and the m's are the various slopes associated with those independent variables. We also still have an intercept, b.

Multiple linear regression is a little harder to visualize and think about because this is no longer a line that can be visualized in two dimensions. It is a linear surface in two, three, or more dimensions. However, many of the same techniques that we used for our single linear regression will carry through.

Multiple linear regression has the same assumptions as regular linear regression. However, there are a few more pitfalls that we should keep in mind:

- **Overfitting**: By adding more and more independent variables to our model, we are increasing our model complexity, which puts us at risk of overfitting. One technique to deal with this problem, which I would recommend looking into, is called **regularization**. Regularization creates a penalty term in your model that is a function of the complexity of your model, which helps keep this effect in check.
- **Relative Scale**: In some cases, one of your independent variables will be orders of magnitude different in scale than another independent variable. The larger of these could wash out any effect of the smaller, and you may need to consider normalizing your variables.

With this in mind, let's try to expand our **Sales** model from a linear regression model to a multiple regression model. Looking back at our scatter plots from the previous section, we can see that **Radio** also appears to be linearly correlated with **Sales**, so let's try to create a multiple linear regression model that looks like the following:

$$Sales = m_1\ TV + m_2\ Radio + b$$

To do this with `github.com/sajari/regression`, we just need to label another variable in the `regression.Regression` value and make sure that these values get paired in the training data points. We will then run the regression and see how the formula comes out:

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
trainingData, err := reader.ReadAll()
```

```
if err != nil {
    log.Fatal(err)
}

// In this case we are going to try and model our Sales
// by the TV and Radio features plus an intercept.
var r regression.Regression
r.SetObserved("Sales")
r.SetVar(0, "TV")
r.SetVar(1, "Radio")

// Loop over the CSV records adding the training data.
for i, record := range trainingData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yVal, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the Radio value.
    radioVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Add these points to the regression value.
    r.Train(regression.DataPoint(yVal, []float64{tvVal, radioVal}))
}

// Train/fit the regression model.
r.Run()

// Output the trained model parameters.
fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)
```

Compiling and running this gives us the following regression formula:

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 2.93 + TV*0.05 + Radio*0.18
```

As you can see, the regression formula now includes an additional term for the `Radio` independent variable. The intercept value has also changed from our previous single regression model.

We can test this model similarly to the single regression model using the `Predict` method:

```
// Open the test dataset file.
f, err = os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV reader reading from the opened file.
reader = csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the test data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yObserved, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
```

```
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the Radio value.
        radioVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted, err := r.Predict([]float64{tvVal, radioVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
```

Running this reveals the following `MAE` for our new multiple regression model:

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 2.93 + TV*0.05 + Radio*0.18

MAE = 1.26
```

Our new multiple regression model has improved our MAE! Now we are definitely in pretty good shape to predict `Sales` based on our advertising spends. You could also try adding `Newspaper` to the model as a follow-up exercise to see how the model performance is influenced.



Remember that as you add more complication to the model, you are sacrificing simplicity and you are potentially in danger of overfitting, so you should only add more complication if the gains in model performance actually create more value for your use case.

Nonlinear and other types of regression

Although we have focused on linear regression in this chapter, you certainly are not limited to performing regression with linear formulas. You can model your dependent variable by one or more nonlinear terms such as powers, exponentials, or other transformations on your independent variables. For example, we could model *Sales* by a polynomial series of *TV* terms:

$$Sales = m_1\ TV + m_2\ TV^2 + m_3\ TV^3 + \ldots + b$$

Keep in mind, however, that as you add this complexity, you are again putting yourself in danger of overfitting.

In terms of implementing non-linear regressions, you cannot use `github.com/sajari/regression`, which is limited to linear regression. However, `go-hep.org/x/hep/fit` allows you to fit or train certain nonlinear models, and there are other various people in the Go community that have, or are, developing other tools for nonlinear modeling.

There are also other linear regression techniques, outside of OLS, that help overcome some of the assumptions and weaknesses associated with least squared linear regression. These include **ridge regression** and **lasso regression**. Both of these techniques penalize regression coefficients so as to mitigate the effects of multicollinearity and non-normality of independent variables.

In terms of Go implementations, ridge regression is implemented in `github.com/berkmancenter/ridge`. As opposed to `github.com/sajari/regression`, our independent variable and dependent variable data is input into `github.com/berkmancenter/ridge` via gonum matrices. Thus, to illustrate this method, let's first form a matrix containing our advertising spend features (`TV`, `Radio`, and `Newspaper`) and a matrix containing our `Sales` data. Note that in `github.com/berkmancenter/ridge`, we need to explicitly add a column to our input independent variable matrix for an intercept if we want to have an intercept in our model. Each value in this column is just `1.0`.

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
```

```
reader.FieldsPerRecord = 4

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// featureData will hold all the float values that will eventually be
// used to form our matrix of features.
featureData := make([]float64, 4*len(rawCSVData))
yData := make([]float64, len(rawCSVData))

// featureIndex and yIndex will track the current index of the matrix
values.
var featureIndex int
var yIndex int

// Sequentially move the rows into a slice of floats.
for idx, record := range rawCSVData {
    // Skip the header row.
    if idx == 0 {
        continue
    }

    // Loop over the float columns.
    for i, val := range record {

        // Convert the value to a float.
        valParsed, err := strconv.ParseFloat(val, 64)
        if err != nil {
            log.Fatal(err)
        }

        if i < 3 {

            // Add an intercept to the model.
            if i == 0 {
                featureData[featureIndex] = 1
                featureIndex++
            }

            // Add the float value to the slice of feature floats.
            featureData[featureIndex] = valParsed
            featureIndex++
        }

        if i == 3 {
```

```
            // Add the float value to the slice of y floats.
            yData[yIndex] = valParsed
            yIndex++
        }
    }
}

// Form the matrices that will be input to our regression.
features := mat64.NewDense(len(rawCSVData), 4, featureData)
y := mat64.NewVector(len(rawCSVData), yData)
```

Next, we create a new `ridge.RidgeRegression` value with our independent and dependent variable matrices and call the `Regress()` method to train our model. We can then print out our trained regression formula:

```
// Create a new RidgeRegression value, where 1.0 is the
// penalty value.
r := ridge.New(features, y, 1.0)

// Train our regression model.
r.Regress()

// Print our regression formula.
c1 := r.Coefficients.At(0, 0)
c2 := r.Coefficients.At(1, 0)
c3 := r.Coefficients.At(2, 0)
c4 := r.Coefficients.At(3, 0)
fmt.Printf("\nRegression formula:\n")
fmt.Printf("y = %0.3f + %0.3f TV + %0.3f Radio + %0.3f Newspaper\n\n", c1,
c2, c3, c4)
```

Compiling this program and running it gives the following regression formula:

```
$ go build
$ ./myprogram

Regression formula:
y = 3.038 + 0.047 TV + 0.177 Radio + 0.001 Newspaper
```

Here, you can that see the coefficients for `TV` and `Radio` are similar to what we got with least squares regression, but they are slightly different. Also, note that we went ahead and added a term for the `Newspaper` feature.

We can test this ridge regression formula by creating our own `predict` function:

```
// predict uses our trained regression model to made a prediction based on
a
// TV, Radio, and Newspaper value.
func predict(tv, radio, newspaper float64) float64 {
        return 3.038 + tv*0.047 + 0.177*radio + 0.001*newspaper
}
```

Then, we use this `predict` function to test our ridge regression formula on our test examples:

```
// Open the test dataset file.
f, err := os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the holdout data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yObserved, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
```

```
        }

        // Parse the Radio value.
        radioVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the Newspaper value.
        newspaperVal, err := strconv.ParseFloat(record[2], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted := predict(tvVal, radioVal, newspaperVal)

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("\nMAE = %0.2f\n\n", mAE)
```

Compiling and running this gives us the following new `MAE`:

```
$ go build
$ ./myprogram

MAE = 1.26
```

Notice that adding `Newspaper` to the model did not actually improve our `MAE`. Thus, this would not be a good idea in this case, because it is adding further complications and not providing any significant changes in our model performance.



Any complication or sophistication that you are adding to a model should be accompanied by a measurable justification for this added complication. Using a sophisticated model because it is intellectually interesting is a recipe for headaches.

References

Linear regression:

- Ordinary least squares regression explained visually: `http://setosa.io/ev/ordinary-least-squares-regression/`
- `github.com/sajari/regression` docs: `http://godoc.org/github.com/sajari/regression`

Multiple regression:

- Multiple regression visualization: `http://shiny.stat.calpoly.edu/3d_regression/`

Nonlinear and other regressions:

- `go-hep.org/x/hep/fit` docs: `https://godoc.org/go-hep.org/x/hep/fit`
- `github.com/berkmancenter/ridge` docs: `https://godoc.org/github.com/berkmancenter/ridge`

Summary

Congratulations! You have officially done machine learning using Go. In particular, you have learned about regression models, including linear regression, multiple regression, nonlinear regression, and ridge regression. You should be able to implement basic linear regressions and multiple regressions in Go.

Now that we have our feet wet with machine learning, we are going to move on to classification problems in the next chapter.

5
Classification

When many people think about machine learning or artificial intelligence, they probably first think about machine learning to solve classification problems. These are problems where we want to train a model to predict one of a finite number of distinct categories. For example, we may want to predict if a financial transaction is fraudulent or not fraudulent, or we may want to predict whether an image contains a hot dog, airplane, cat, and so on, or none of those things.

The categories that we try to predict could number from two to many hundreds or thousands. In addition, we could be making our predictions based on only a few attributes or many attributes. All of the scenarios arising from these combinations lead to a host of models with a corresponding host of assumptions, advantages, and disadvantages.

We will cover some of these models in this chapter and later in the book, but there are many that we will skip for brevity. However, as with any of the problems that we tackle in this book, simplicity and integrity should be a major concern as we choose a type of model for our use cases. There are highly sophisticated and complicated models that solve certain problems very well, but these models are not necessary for many use cases. Applying simple and interpretable classification models should continue to be one of our goals.

Understanding classification model jargon

As with regression, classification problems come with their own set of jargon. There is some overlap with terms used in regression, but there are also some new terms specific to classification:

- **Categories**, **labels**, or **classes**: These terms are used interchangeably to represent the various distinct choices for our prediction. For example, we could have a fraud class and a not fraud class, or we could have sitting, standing, running, and walking categories.
- **Binary classification**: This type of classification is one with only two categories or classes, such as yes/no or fraud/not fraud.
- **Multi-class classification**: This type of classification is one with more than two classes, such as a classification trying to assign one of hot dog, airplane, cat, and so on, to an image.
- **Labeled data** or **annotated data**: Real-world observations or records that have been paired with their corresponding class. For example, if we are predicting fraud via transaction time, this data would include a bunch of measured transaction times along with a corresponding label indicating whether they were fraudulent or not.

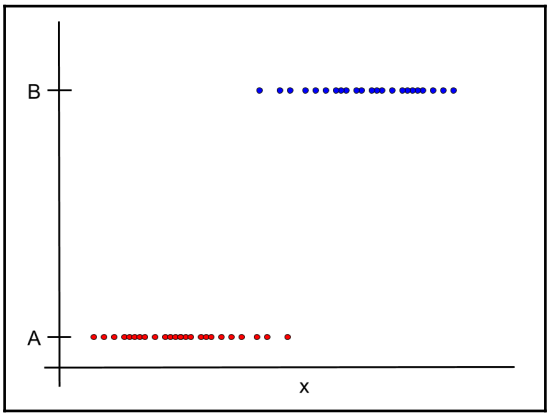
Logistic regression

The first classification model that we are going to explore is called **logistic regression**. As you can tell from the name, this method is based on a regression, which we discussed in more detail in the previous chapter. However, this particular regression uses a function that is particularly well suited to classification problems.

This is also a simple and interpretable model, which makes it a great first choice when solving classification problems. There are a variety of existing Go packages that implement logistic regression for you, including `github.com/xlvector/hector`, `github.com/cdipaolo/goml`, and `github.com/sjwhitworth/golearn`. However, in our example, we will implement logistic regression from scratch, so that you can both form a full understanding of what goes into training a model and understand the simplicity of logistic regression. Further, in some cases, you may want to utilize a from-scratch implementation as illustrated in the following section to avoid extra dependencies in your code base.

Overview of logistic regression

Let's say that we have two classes, *A* and *B*, that we are trying to predict. Let's also suppose that we are trying to predict *A* or *B* based on a variable *x*. Classes *A* and *B* might look something like this when plotted against *x*:



Although we could draw a line modeling this behavior, this is clearly not linear behavior and does not fit the assumptions of linear regression. The shape of the data is more of a step from one class to another class as a function of *x*. What we really need is some function that goes to and stays at *A* for low values of *x*, and goes to and stays at *B* for higher values of *x*.

Well, we are in luck! There is such a function. The function is called the **logistic function**, and it gives logistic regression its name. It has the following form:

$$f(x) = \frac{1}{1 + e^{-x}}$$

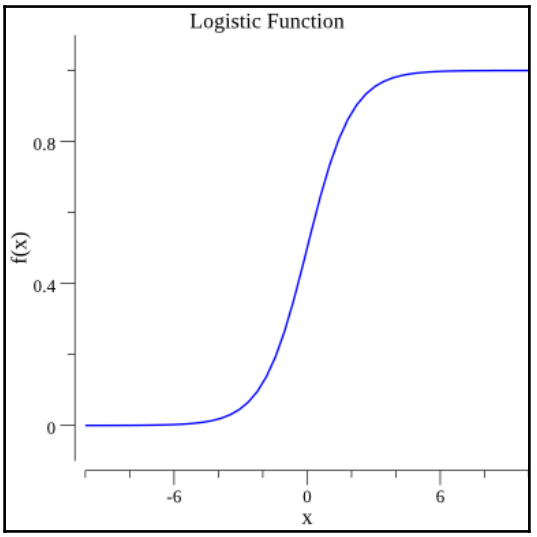
Implemented in Go, this looks as follows:

```
// logistic implements the logistic function, which
// is used in logistic regression.
func logistic(x float64) float64 {
        return 1 / (1 + math.Exp(-x))
}
```

Let's plot the logistic function with `gonum.org/v1/plot` to see what it looks like:

```
// Create a new plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.Title.Text = "Logistic Function"
p.X.Label.Text = "x"
p.Y.Label.Text = "f(x)"

// Create the plotter function.
logisticPlotter := plotter.NewFunction(func(x float64) float64 { return
logistic(x) })
logisticPlotter.Color = color.RGBA{B: 255, A: 255}

// Add the plotter function to the plot.
p.Add(logisticPlotter)

// Set the axis ranges.  Unlike other data sets,
// functions don't set the axis ranges automatically
// since functions don't necessarily have a
// finite range of x and y values.
p.X.Min = -10
p.X.Max = 10
p.Y.Min = -0.1
p.Y.Max = 1.1

// Save the plot to a PNG file.
if err := p.Save(4*vg.Inch, 4*vg.Inch, "logistic.png"); err != nil {
    log.Fatal(err)
}
```

Compiling and running this plotting code creates the following graph:



As you can see, this function has the step-like behavior that we are looking for to model the steps between classes *A* and *B* (imagining that *A* corresponds to *0.0* and *B* corresponds to *1.0*).

Not only that, the logistic function has some really convenient properties that we can take advantage of while doing classification. To see this, let's take a step back and consider how we might model *p*, the probability of one of the classes *A* or *B*, occurring. One way to do this would be to model the *log* of the **odds ratio**, *log(p / (1 - p))* linearly, where the odds ratio tells us how the presence or absence of class *A* has an effect on the presence or absence of class *B*. The reason for using this strange *log* (referred to as **logit**) will make sense shortly, but for now, let's just assume that we want to model this linearly as follows:

$$\log\left(\frac{p}{1-p}\right) = mx + b$$

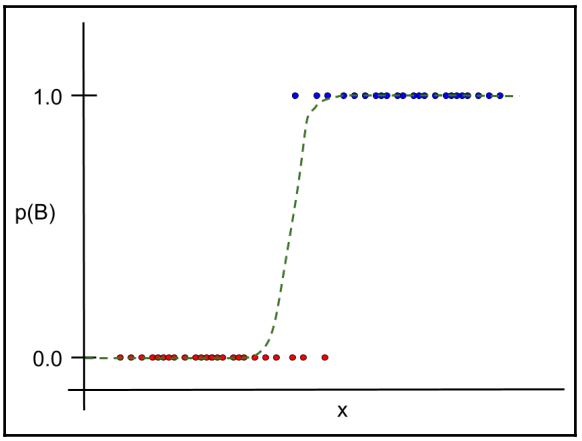
Now, if we take the exponential of this odds ratio, we get the following:

$$\frac{p}{1-p} = e^{mx+b}$$

This is what we get when we simplify the preceding equation:

$$p = \frac{1}{1 + e^{-mx-b}}$$

If you look at the right-hand side of this equation, you will see that our logistic function has popped up. This equation then gives some formal underpinning to our assumption that the logistic function would be good for modeling the separation between two classes: *A* and *B*. If, for instance, we take *p* to be the probability of observing *B* and we fit the logistic function to our data, we could get a model that predicts the probability of *B* as a function of *x* (and thus one minus the probability of predicting *A*). This is pictured in the following plot, in which we have formalized our original plot of *A* and *B* and overlaid the logistic function that models probability:



Thus, creating a logistic regression model involves finding the logistic function that maximizes the number of observations that we can predict with the logistic function.



Note that one advantage of logistic regression is that it remains simple and interpretable. However, the coefficients *m* and *b* in the model do not have the same interpretation as in linear regression. The coefficient *m* (or coefficients m_1, m_2, and so on, if we have multiple independent variables) has an exponential relationship with the odds ratio. Thus, if you have a coefficient *m* of *0.5*, this is related to the odds ratio via *exp(0.5 x)*. If we had two coefficients $exp(0.5\ x_1 + 1.0\ x_2)$, we could conclude that the odds of the modeled class for x_1 is *exp(0.5) = 1.65* when compared to *exp(1.0) = 2.72* for x_2. In other words, we cannot directly compare the coefficients. We need to keep them in the context of the exponential.

Logistic regression assumptions and pitfalls

Remember the long list of assumptions that applied to linear regression? Well, logistic regression is not limited by those same assumptions. However, there are still some important assumptions that we make when using logistic regression:

- **Linear relationship with the log odds**: As we discussed earlier, underlying logistic regression is an assumption that we can model the log of the odds ratio with a line.
- **Encoding of dependent variable**: When we set up our model earlier, we assumed that we were trying to predict the probability of *B*, where a probability of 1.0 corresponded to a positive *B* example. Thus, we need to prepare our data with this type of encoding. This will be demonstrated in the following example.
- **Independence of observations**: Each of the examples of *x* in our data must be independent. That is, we have to avoid things like including the same example several times.

Also, some common pitfalls of logistic regression to keep in mind are as follows:

- Logistic regression can be more sensitive to outliers than other classification techniques. Keep this in mind and try to profile your data accordingly.
- As logistic regression relies on an exponential function that never truly goes to *0.0* or *1.0* (except at +/- infinity), you may have very small degradations in your evaluation metric.

All being said, logistic regression is a fairly robust method that remains interpretable. It is a flexible model that should be at the top of your list when considering how you might solve classification problems.

Logistic regression example

The dataset that we are going to use to illustrate logistic regression is data corresponding to loans issues by LendingClub. LendingClub publishes this data quarterly and it can be found in its original form at `https://www.lendingclub.com/info/download-data.action`. We will work with a trimmed down and simplified version of this data (available in the code bundles that come with this book) including only two columns, `FICO.Range` (indicating a loan applicants credit score as given by Fair, Isaac and Company, or FICO) and `Interest.Rate` (indicating the interest rate of a loan granted to the loan applicant). The data looks like this:

```
$ head loan_data.csv
```

```
FICO.Range,Interest.Rate
735-739,8.90%
715-719,12.12%
690-694,21.98%
695-699,9.99%
695-699,11.71%
670-674,15.31%
720-724,7.90%
705-709,17.14%
685-689,14.33%
```

Our goal for this exercise will be to create a logistic regression model that will tell us, for a given credit score, if we can get a loan at or below a certain interest rate. For example, let's say that we are interested in getting an interest rate below 12%. Our model would tell us that we could (yes, or class one) or could not (no, class two) get a loan, given a certain credit score.

Cleaning and profiling the data

Looking at the preceding sample of the loan data, we can see that it is not exactly in the form that we need for our classification. Specifically, we need to do the following:

1. Remove non-numerical characters from the interest rate and FICO score columns.
2. Encode our interest rate into two classes for a given interest rate threshold. We will use *1.0* to represent our first class (yes, we can get the loan with that interest rate) and *0.0* to represent our second class (no, we cannot get the loan with that interest rate).
3. Select a single value for the FICO credit score. We are given a range of credit scores, but we need a single value. The average, minimum, or maximum score are natural choices and, in our example, we will use the minimum value (to be conservative).
4. In this case, we are going to **standardize** our FICO scores (by subtracting the minimum score value from each score and then dividing by the score range). This will spread out our score values between *0.0* and *1.0*. We need to have a justification for this as it is making our data less readable. However, there is a good justification. We will be training our logistic regression using a gradient descent method that can perform better with normalized data. In fact, when running the same example with non-normalized data, there are convergence issues.

Let's write a Go program that will clean our data for a given interest rate (12% for our example). We will read in the data from a given file, parse the values using `encoding/csv`, and put the cleaned data in an output file called `clean_loan_data.csv`. During the cleaning of the data we will make use of the following minimum and maximum values, that we define as constants:

```
const (
    scoreMax = 830.0
    scoreMin = 640.0
)
```

Then, the actual cleaning functionality is shown in the following code:

```
// Open the loan dataset file.
f, err := os.Open("loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 2

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Create the output file.
f, err = os.Create("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV writer.
w := csv.NewWriter(f)

// Sequentially move the rows writing out the parsed values.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {
```

```
            // Write the header to the output file.
            if err := w.Write(record); err != nil {
                log.Fatal(err)
            }
            continue
        }

        // Initialize a slice to hold our parsed values.
        outRecord := make([]string, 2)

        // Parse and standardize the FICO score.
        score, err := strconv.ParseFloat(strings.Split(record[0], "-")[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        outRecord[0] = strconv.FormatFloat((score-scoreMin)/(scoreMax-
    scoreMin), 'f', 4, 64)

        // Parse the Interest rate class.
        rate, err := strconv.ParseFloat(strings.TrimSuffix(record[1], "%"), 64)
        if err != nil {
            log.Fatal(err)
        }

        if rate <= 12.0 {
            outRecord[1] = "1.0"

            // Write the record to the output file.
            if err := w.Write(outRecord); err != nil {
                log.Fatal(err)
            }
            continue
        }

        outRecord[1] = "0.0"

        // Write the record to the output file.
        if err := w.Write(outRecord); err != nil {
            log.Fatal(err)
        }
    }

    // Write any buffered data to the underlying writer (standard output).
    w.Flush()

    if err := w.Error(); err != nil {
        log.Fatal(err)
```

```
}
```

Compiling this and running it confirms our desired output:

```
$ go build
$ ./example3
$ head clean_loan_data.csv
FICO_score,class
0.5000,1.0
0.3947,0.0
0.2632,0.0
0.2895,1.0
0.2895,1.0
0.1579,0.0
0.4211,1.0
0.3421,0.0
0.2368,0.0
```

Great! We have our data in the desired format. Now, let's gain a little more intuition about our data by creating histograms for the FICO score and interest rate data and calculating summary statistics. We will utilize `github.com/kniren/gota/dataframe` to calculate summary statistics and `gonum.org/v1/plot` to generate histograms:

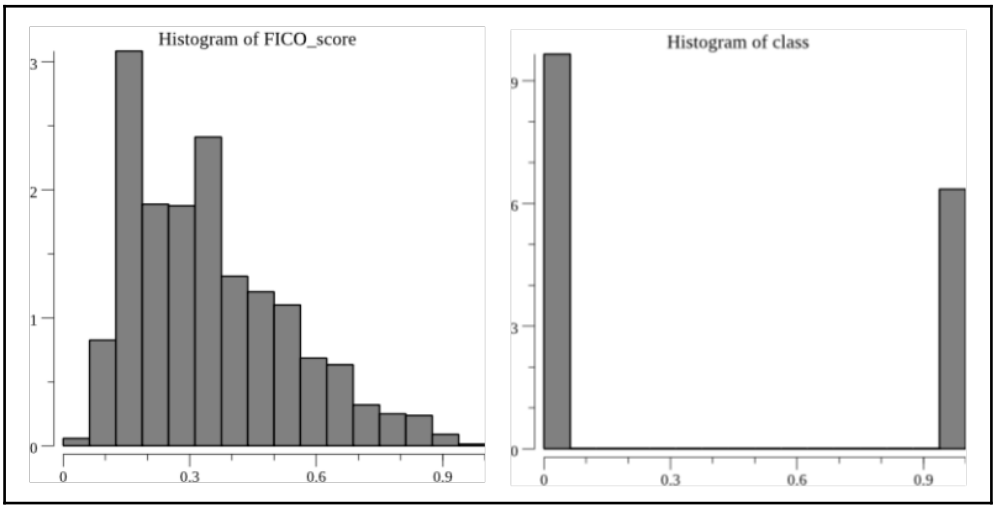
```
// Open the CSV file.
loanDataFile, err := os.Open("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer loanDataFile.Close()

// Create a dataframe from the CSV file.
loanDF := dataframe.ReadCSV(loanDataFile)

// Use the Describe method to calculate summary statistics
// for all of the columns in one shot.
loanSummary := loanDF.Describe()

// Output the summary statistics to stdout.
fmt.Println(loanSummary)

// Create a histogram for each of the columns in the dataset.
for _, colName := range loanDF.Names() {

    // Create a plotter.Values value and fill it with the
    // values from the respective column of the dataframe.
    plotVals := make(plotter.Values, loanDF.Nrow())
    for i, floatVal := range loanDF.Col(colName).Float() {
        plotVals[i] = floatVal
```

```
        }

        // Make a plot and set its title.
        p, err := plot.New()
        if err != nil {
            log.Fatal(err)
        }
        p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

        // Create a histogram of our values.
        h, err := plotter.NewHist(plotVals, 16)
        if err != nil {
            log.Fatal(err)
        }

        // Normalize the histogram.
        h.Normalize(1)

        // Add the histogram to the plot.
        p.Add(h)

        // Save the plot to a PNG file.
        if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err != nil
{
            log.Fatal(err)
        }
}
```

Running this results in the following output:

```
$ go build
$ ./myprogram
[7x3] DataFrame

    column FICO_score class
 0: mean 0.346782 0.396800
 1: stddev 0.184383 0.489332
 2: min 0.000000 0.000000
 3: 25% 0.210500 0.000000
 4: 50% 0.315800 0.000000
 5: 75% 0.447400 1.000000
 6: max 1.000000 1.000000
    <string> <float> <float>

$ ls *.png
class_hist.png FICO_score_hist.png
```

We can see that the average credit score is pretty high at *706.1* and there is a pretty good balance between the classes one and zero, as indicated by a mean near *0.5*. However, there appears to be more class zero examples (which corresponds to not receiving a loan with an interest rate of 12% or lower). Also, the `*.png` histogram graphs look as follows:



This confirms our suspicions about the balance between the classes and shows us that the FICO scores are skewed a bit to lower values.

Creating our training and test sets

Similar to our examples in the previous chapter, we need to split our data into a training and test set. We will once again use `github.com/kniren/gota/dataframe` to do this:

```
// Open the clean loan dataset file.
f, err := os.Open("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
loanDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * loanDF.Nrow()) / 5
testNum := loanDF.Nrow() / 5
if trainingNum+testNum < loanDF.Nrow() {
```

```
        trainingNum++
    }

    // Create the subset indices.
    trainingIdx := make([]int, trainingNum)
    testIdx := make([]int, testNum)

    // Enumerate the training indices.
    for i := 0; i < trainingNum; i++ {
        trainingIdx[i] = i
    }

    // Enumerate the test indices.
    for i := 0; i < testNum; i++ {
        testIdx[i] = trainingNum + i
    }

    // Create the subset dataframes.
    trainingDF := loanDF.Subset(trainingIdx)
    testDF := loanDF.Subset(testIdx)

    // Create a map that will be used in writing the data
    // to files.
    setMap := map[int]dataframe.DataFrame{
        0: trainingDF,
        1: testDF,
    }

    // Create the respective files.
    for idx, setName := range []string{"training.csv", "test.csv"} {

        // Save the filtered dataset file.
        f, err := os.Create(setName)
        if err != nil {
            log.Fatal(err)
        }

        // Create a buffered writer.
        w := bufio.NewWriter(f)

        // Write the dataframe out as a CSV.
        if err := setMap[idx].WriteCSV(w); err != nil {
            log.Fatal(err)
        }
    }
```

Compiling and running this results in two files with our training and test examples:

```
$ go build
$ ./myprogram
$ wc -l *.csv
 2046 clean_loan_data.csv
  410 test.csv
 1638 training.csv
 4094 total
```

Training and testing the logistic regression model

Now, let's create a function that trains a logistic regression model. This function needs to perform the following:

1. Accept our FICO score data as an independent variable.
2. Add an intercept to our model.
3. Initialize and optimize the coefficients (or weights) of the logistic regression model.
4. Return the optimized weights which define our trained model.

To optimize the coefficient/weights, we will use a technique called **stochastic gradient descent**. This technique will be covered in greater detail in the Appendix, *Algorithms/Techniques Related to Machine Learning*. For now, suffice it to say that we are trying to make predictions with some non-optimized weights, calculating an error for those weights, and then updating them iteratively to maximize the likelihood of making a correct prediction.

An implementation of this optimization is as follows. The function takes the following as input:

- `features`: A pointer to a gonum `mat64.Dense` matrix. This matrix includes a column for any independent variable that we are using (FICO score, in our case) along with columns of 1.0s representing an intercept.
- `labels`: A slice of floats including all of the class labels corresponding to our `features`.
- `numSteps`: A maximum number of iterations for the optimization.
- `learningRate`: An adjustable parameter that helps with the convergence of the optimization.

The function then outputs the optimized weights for the logistic regression model:

```
// logisticRegression fits a logistic regression model
// for the given data.
func logisticRegression(features *mat64.Dense, labels []float64, numSteps
int, learningRate float64) []float64 {

        // Initialize random weights.
        _, numWeights := features.Dims()
        weights := make([]float64, numWeights)

        s := rand.NewSource(time.Now().UnixNano())
        r := rand.New(s)

        for idx, _ := range weights {
                weights[idx] = r.Float64()
        }

        // Iteratively optimize the weights.
        for i := 0; i < numSteps; i++ {

        // Initialize a variable to accumulate error for this iteration.
        var sumError float64

        // Make predictions for each label and accumulate error.
        for idx, label := range labels {

            // Get the features corresponding to this label.
            featureRow := mat64.Row(nil, idx, features)

            // Calculate the error for this iteration's weights.
            pred := logistic(featureRow[0]*weights[0]
            featureRow[1]*weights[1])
            predError := label - pred
            sumError += math.Pow(predError, 2)

            // Update the feature weights.
            for j := 0; j < len(featureRow); j++ {
                weights[j] += learningRate * predError * pred * (1 - pred)
* featureRow[j]
            }
        }
    }

    return weights
}
```

As you can see, this function is relatively compact and simple. This will keep our code readable and allow people on our team to quickly understand what is happening in our model without hiding things in a black box.



Despite the popularity of R and Python in machine learning, you can see that machine learning algorithms can be implemented quickly and compactly in Go. Moreover, these implementations immediately achieve a level of integrity that far surpasses naive implementations in other languages.

To train our logistic regression model on our training dataset, we will parse our training file with `encoding/csv` and then supply the necessary parameters to our `logisticRegression` function. This process is as follows, along with some code to output our trained logistic formula to `stdout`:

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 2

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// featureData and labels will hold all the float values that
// will eventually be used in our training.
featureData := make([]float64, 2*len(rawCSVData))
labels := make([]float64, len(rawCSVData))

// featureIndex will track the current index of the features
// matrix values.
var featureIndex int

// Sequentially move the rows into the slices of floats.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {
```

```
            continue
        }

        // Add the FICO score feature.
        featureVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        featureData[featureIndex] = featureVal

        // Add an intercept.
        featureData[featureIndex+1] = 1.0

        // Increment our feature row.
        featureIndex += 2

        // Add the class label.
        labelVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Fatal(err)
        }

        labels[idx] = labelVal
    }

    // Form a matrix from the features.
    features := mat64.NewDense(len(rawCSVData), 2, featureData)

    // Train the logistic regression model.
    weights := logisticRegression(features, labels, 100, 0.3)

    // Output the Logistic Regression model formula to stdout.
    formula := "p = 1 / ( 1 + exp(- m1 * FICO.score - m2) )"
    fmt.Printf("\n%s\n\nm1 = %0.2f\nm2 = %0.2f\n\n", formula, weights[0],
    weights[1])
```

Compiling and running this training functionality results in the following trained logistic regression formula:

```
$ go build
$ ./myprogram

p = 1 / ( 1 + exp(- m1 * FICO.score - m2) )

m1 = 13.65
m2 = -4.89
```

We can then utilize this formula directly to make predictions. Remember, however, that this model makes a prediction for the probability of getting the loan (at the interest rate of 12%). As such, we need to utilize a threshold for the probability in making predictions. For example, we can say that any *p* of *0.5+* will be deemed positive (class one, or getting the loan) and any lower *p* values will be deemed negative. This type of prediction is implemented in the following function:

```
// predict makes a prediction based on our
// trained logistic regression model.
func predict(score float64) float64 {

    // Calculate the predicted probability.
    p := 1 / (1 + math.Exp(-13.65*score+4.89))

    // Output the corresponding class.
    if p >= 0.5 {
        return 1.0
    }

    return 0.0
}
```

Using this `predict` function, we can evaluate our trained logistic regression model using one of the evaluation metrics introduced earlier in the book. In this case, let's use accuracy, as shown in the following code:

```
// Open the test examples.
f, err := os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values
// form the labeled data file.
var observed []float64
var predicted []float64

// line will track row numbers for logging.
line := 1

// Read in the records looking for unexpected types in the columns.
for {
```

```
        // Read in a row. Check if we are at the end of the file.
        record, err := reader.Read()
        if err == io.EOF {
            break
        }

        // Skip the header.
        if line == 1 {
            line++
            continue
        }
        // Read in the observed value.
        observedVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Printf("Parsing line %d failed, unexpected type\n", line)
            continue
        }

        // Make the corresponding prediction.
        score, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Printf("Parsing line %d failed, unexpected type\n", line)
            continue
        }

        predictedVal := predict(score)

        // Append the record to our slice, if it has the expected type.
        observed = append(observed, observedVal)
        predicted = append(predicted, predictedVal)
        line++
    }

    // This variable will hold our count of true positive and
    // true negative values.
    var truePosNeg int

    // Accumulate the true positive/negative count.
    for idx, oVal := range observed {
        if oVal == predicted[idx] {
            truePosNeg++
        }
    }

    // Calculate the accuracy (subset accuracy).
    accuracy := float64(truePosNeg) / float64(len(observed))

    // Output the Accuracy value to standard out.
```

```
fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

Running this test on our data results in the following accuracy:

```
$ go build
$ ./myprogram

Accuracy = 0.83
```

Nice! 83% accuracy is not that bad for a machine learning model that we implemented in about 30 lines of Go. With this simple model, we were able to predict whether, given a certain credit score, loan applicants would be accepted for a loan with less than or equal to 12% interest. Not only that, we did it with real-world messy data from a real company.

k-nearest neighbors

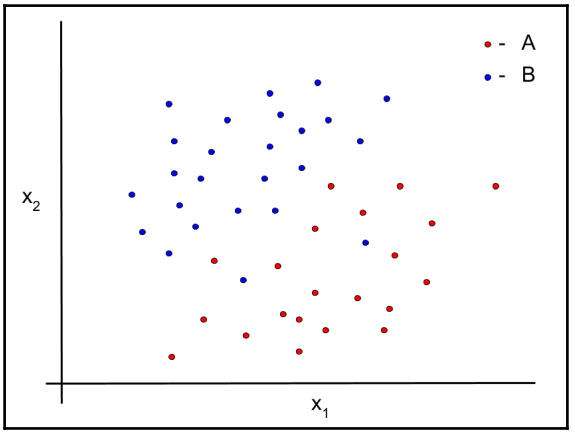
Moving on from logistic regression, let's try our first non-regression model, **k-nearest neighbors** (**kNN**). kNN is also a simple classification model, and it's one of the easiest model algorithms to grasp. It follows on from the basic premise that if I want to classify a record, I should consider other similar records.

kNN is implemented in multiple existing Go packages including `github.com/sjwhitworth/golearn`, `github.com/rikonor/go-ann`, `github.com/akreal/knn`, and `github.com/cdipaolo/goml`. We will be using the `github.com/sjwhitworth/golearn` implementation, which will serve as a great introduction to using `github.com/sjwhitworth/golearn` in general.

Overview of kNN

As mentioned, kNN operates on the principle that we should classify records according to similar records. There are some details to be dealt with in defining similar. However, kNN does not have the complexity of parameters and options that come with many models.

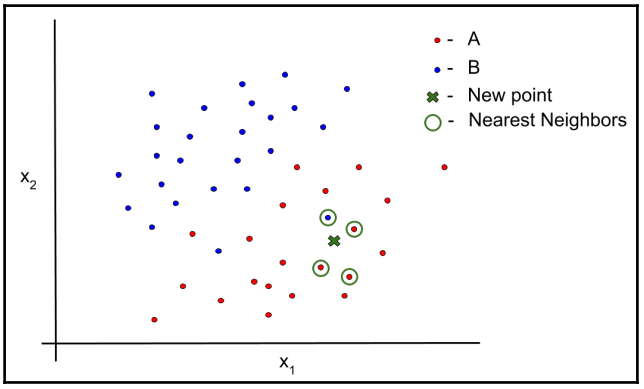
Imagine again that we have two classes *A* and *B*. This time, however, suppose that we are wanting to classify based on two features, x_1 and x_2. Visually, this looks something like the following:



Now, suppose we have a new point of data with an unknown class. This new point of data will sit somewhere in this space. kNN says that, to classify this new point, we should perform the following:

1. Find the *k* nearest point to the new point according to some measure of nearness (straight distance in this space of x_1 and x_2, for example).
2. Determine how many of those *k* nearest neighbors are of class *A* and how many are of class *B*.
3. Classify the new point as the dominant class among the *k* nearest neighbors.

For example, if we choose *k* to be four, this would look something as follows:



Our mystery point has three *A* nearest neighbors and only one *B* nearest neighbor. Thus, we would classify the new mystery point as a class *A* point.

There are many measures of similarity that you can use to determine the *k* nearest neighbors. The most common of these is the Euclidean distance, which is just the straight line distance from one point to the next in the space made up of your features (x_1 and x_2 in our example). Others include **Manhattan distance**, **Minkowski distance**, **cosine similarity**, and **Jaccard Similarity**.



As with evaluation metrics, there are a whole host of ways that distance or similarity can be measured. When using kNN, you should look into the advantages and disadvantages of these measures and choose one that fits your use case and data. When in doubt, however, you can try starting with the Euclidean distance.

kNN assumptions and pitfalls

Due to its simplicity, kNN does not have too many assumptions. However, there are some common pitfalls that you should be aware of as you apply kNN:

- kNN is evaluated lazily. By this, we mean that the distances or similarities are calculated when we need to make a prediction. There is not really anything to train or fit prior to making a prediction. This has some advantages, but the calculation and search over points can be slow when you have many data points.
- The choice of *k* is up to you, but you should put some formalism around choosing *k* and provide justification for the *k* that you choose. A common technique to choose *k* is just to search over a range of *k* values. You could, for example, start with $k = 2$. Then, you could start increasing *k*, and for each *k*, evaluate on a test set.
- kNN does not take into consideration which features are more important than other features. Moreover, if the scale of certainty of your features is much larger than other features, this could unnaturally weight the importance of those larger features.

kNN example

For this, and the remaining examples in this chapter, we are going to solve a classic classification problem using a dataset about **iris flowers**. The dataset looks like this:

```
$ head iris.csv
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
```

The first four columns are various measurements of iris flowers and the last column is a corresponding species label. The goal of this example will be to create a kNN classifier that is able to predict the species of an iris flower from a set of measurements. There are three species of flowers, or three classes, making this a multi-class classification (in contrast to the binary classification that we did with logistic regression).



You may remember that we already profiled the iris dataset in detail in `Chapter 2`, *Matrices, Probability, and Statistics*. We will not reprofile the data here. However, it is still important that we have intuition about our data as we develop the kNN model. Make sure you flip back to `Chapter 2`, *Matrices, Probability, and Statistics*, to remind yourself about the distributions of variables in this dataset.

We will be using `github.com/sjwhitworth/golearn` in this example. `github.com/sjwhitworth/golearn` implements a variety of machine learning models, including kNN and some others that we will explore shortly. `github.com/sjwhitworth/golearn` also implements cross-validation. We will take advantage of cross-validation here to perform training, testing, and validation, which is convenient and lets us avoid performing a manual split between training and test sets.

To use any of `github.com/sjwhitworth/golearn`'s models, we must first convert the data into a `github.com/sjwhitworth/golearn` internal format called **instances**. For the iris data, we can do this as follows:

```
// Read in the iris data set into golearn "instances".
irisData, err := base.ParseCSVToInstances("iris.csv", true)
if err != nil {
    log.Fatal(err)
```

```
}
```

Then, initializing our kNN model and performing the cross-validation is quick and simple:

```
// Initialize a new KNN classifier.  We will use a simple
// Euclidean distance measure and k=2.
knn := knn.NewKnnClassifier("euclidean", "linear", 2)

// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, knn, 5)
if err != nil {
    log.Fatal(err)
}
```

Finally, we can get the average accuracy across the five folds of the cross-validation and output that accuracy to `stdout`:

```
// Get the mean, variance and standard deviation of the accuracy for the
// cross validation.
mean, variance := evaluation.GetCrossValidatedMetric(cv,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the cross metrics to standard out.
fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)
```

Compiling and running all of this together gives the following output:

```
$ go build
$ ./myprogram
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
KNN: 95.00 % done
Accuracy
0.95 (+/- 0.05)
```

After some benign logging from the package during the cross-validation, we can see that kNN ($k = 2$) was able to predict iris flower species with 95% accuracy!

The next step here would be to try this model out with a variety of different k values. Actually, it would be a good exercise to plot the accuracy versus k values to see which k value gives you the best performance.

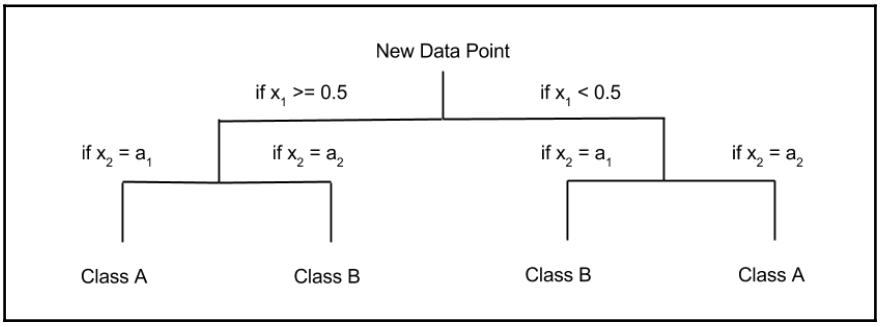
Decision trees and random forests

Tree-based models are very different from the previous types of models that we have discussed, but they are widely utilized and very powerful. You can think about a **decision tree** model like a series of `if-then` statements applied to your data. When you train this type of model, you are constructing a series of control flow statements that eventually allow you to classify records.

Decision trees are implemented in `github.com/sjwhitworth/golearn` and `github.com/xlvector/hector`, among others, and random forests are implemented in `github.com/sjwhitworth/golearn`, `github.com/xlvector/hector`, and `github.com/ryanbressler/CloudForest`, among others. We will utilize `github.com/sjwhitworth/golearn` again in our examples shown in the following section.

Overview of decision trees and random forests

Again, consider our classes *A* and *B*. In this case, suppose that we have one feature, x_1, that ranges from *0.0* to *1.0*, and we have another feature, x_2 that is categorical and can take on one of two values, a_1 and a_2 (this could be something like male/female or red/blue). A decision tree to classify a new data point might look something like the following:



There are a variety of ways to choose how decision trees are constructed, split, and so on. One of the most common ways to determine how decision trees are constructed is using a quantity called **entropy**. This entropy-based approach is discussed in more detail in the *Appendix*, but, basically, we construct the tree and splits based on which features give us the most information about the problem we are solving. The more important features are prioritized higher on the tree.

This prioritization of important features and a natural looking structure makes decision trees very interpretable. This makes decision trees important for applications in which you might have to explain your inferences (for compliance reasons, for example).

However, a single decision tree can be unstable with respect to changes in your training data. In other words, the structure of the tree may change significantly with even small changes in your training data. This can be challenging to manage both operationally and cognitively, and this is one of the reasons why the **random forest** model was created.

A random forest is a collection of decision trees that work together to make a prediction. Random forests are more stable when compared to single decision trees, and they are more robust against overfitting. In fact, this idea of combining models in an **ensemble** is prevalent throughout machine learning both to improve the performance of simple classifiers (such as decision trees) and to help prevent overfitting.

To construct a random forest, we choose N random subsets of features and construct N separate decision trees based on these subsets. When making a prediction, we can then have each of these N decision trees make a prediction. To obtain a final prediction, we can take the majority vote of these N predictions.

Decision tree and random forest assumptions and pitfalls

Tree-based approaches are non-statistical approaches without many of the assumptions that come along with things like regression. However, there are some pitfalls to keep in mind:

- Single decision tree models can easily overfit to your data, especially if you do not limit the depth of the trees. Most implementations allow you to limit this depth via a parameter (or **prune** your decision trees). A pruning parameter will often allow you to remove sections of the tree that have little influence on the predictions, and, thus, reduce the overall complexity of the model.
- When we start talking about ensemble models, such as random forest, we are getting into models that are somewhat opaque. It's very hard to gain intuition about an ensemble of models and you have to treat it like a black box to some degree. Less interpretable models like these should be applied only when necessary.
- Although decision trees themselves are very computationally efficient, random forests can be very computationally inefficient depending on how many features you have and how many trees are in your random forest.

Decision tree example

We will again utilize the iris dataset for this example. You have already learned how to handle this dataset in `github.com/sjwhitworth/golearn`, and we can follow a similar pattern again. We will again use cross-validation. However, this time we will fit a decision tree model:

```
// Read in the iris data set into golearn "instances".
irisData, err := base.ParseCSVToInstances("iris.csv", true)
if err != nil {
    log.Fatal(err)
}

// This is to seed the random processes involved in building the
// decision tree.
rand.Seed(44111342)

// We will use the ID3 algorithm to build our decision tree.  Also, we
// will start with a parameter of 0.6 that controls the train-prune split.
tree := trees.NewID3DecisionTree(0.6)

// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, tree, 5)
if err != nil {
    log.Fatal(err)
}

// Get the mean, variance and standard deviation of the accuracy for the
// cross validation.
mean, variance := evaluation.GetCrossValidatedMetric(cv,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the cross metrics to standard out.
fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)
```

Compiling and running this decision tree model gives the following:

```
$ go build
$ ./myprogram

Accuracy
0.94 (+/- 0.06)
```

94% accuracy this time. Slightly worse than our kNN model, but still very respectable.

Random forest example

`github.com/sjwhitworth/golearn` also implements random forests. To utilize random forest in solving the iris problem, we simply swap our decision tree model for the random forests. We will need to tell the package how many trees we want to build and with how many randomly chosen features per tree.

A sane default for the number of features per tree is the square root of the number of total features, which in our case would be two. We will see that this choice for our small dataset does not produce good results because we need all of the features to make a good prediction here. However, we will illustrate random forest with the sane default to see how it works:

```
// Assemble a random forest with 10 trees and 2 features per tree,
// which is a sane default (number of features per tree is normally set
// to sqrt(number of features)).
rf := ensemble.NewRandomForest(10, 2)

// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, rf, 5)
if err != nil {
    log.Fatal(err)
}
```

Running this gives an accuracy that is worse than the single decision tree. If we change the number of features per tree back up to four, we will recreate the accuracy of the single decision tree. This means that every tree is being trained with the same information as the single decision tree, and thus produces the same results.

Random forest is likely to overkill here and cannot be justified with any performance gains, so it would be best to stick with the single decision tree. This single decision tree is also more interpretable and more efficient.

Naive bayes

The final model that we will cover here for classification is called **Naive bayes**. In `Chapter 2`, *Matrices, Probability, and Statistics*, we discussed the Bayes rule, which forms the basis of this technique. Naive Bayes is a probability-based method like logistic regression, but its basic ideas and assumptions are different.

Naive Bayes is also implemented in `github.com/sjwhitworth/golearn`, which will allow us to easily try it out. However, there are a variety of other Go implementations including `github.com/jbrukh/bayesian`, `github.com/lytics/multibayes`, and `github.com/cdipaolo/goml`.

Overview of naive bayes and its big assumption

Naive bayes operates under one large assumption. This assumption says that the probability of the classes and the presence or absence of a certain feature in our dataset is independent of the presence or absence of other features in our dataset. This allows us to write a very simple formula for the probability of a certain class, given the presence or absence of certain features.

Let's take an example to make this more concrete. Again, let's say that we are trying to predict two classes of emails, *A* and *B* (spam and not spam, for example), based on words in the emails. Naive bayes would assume that the presence/absence of a certain word is independent of other words. If we make this assumption, the probability that a certain class contains certain words is proportional to all of the individual conditional probabilities multiplied together. Using this, the Bayes rule, some chain rules, and our independent assumptions, we can write the conditional probability of a certain class as follows:

$$P\left(B \mid word_1, word_2\right) \propto P\left(B\right) P\left(word_1 \mid B\right) P\left(word_2 \mid B\right)$$

Everything on the right-hand side we can calculate by counting the occurrences of the features and labels in our training set, which is what is done when training the model. Predictions can then be made by stringing together chains of these probabilities.



Actually, in practice, a little trick is used to avoid having to string together a bunch of numbers that are close to zero. We can take the log of the probabilities, add these, and then take the exponential of the expression. This process is generally better in practice.

Naive bayes example

Circling back to our loan dataset for a final time, we are going to use `github.com/sjwhitworth/golearn` to solve the same loan acceptance problem with Naive bayes. We will utilize the same training and test sets that we used in the logistic regression example. However, we need to convert the labels in the datasets to a binary classifier format used in `github.com/sjwhitworth/golearn`. We can write a simple function to perform this conversion:

```
// convertToBinary utilizes built in golearn functionality to
// convert our labels to a binary label format.
func convertToBinary(src base.FixedDataGrid) base.FixedDataGrid {
    b := filters.NewBinaryConvertFilter()
    attrs := base.NonClassAttributes(src)
    for _, a := range attrs {
        b.AddAttribute(a)
    }
    b.Train()
    ret := base.NewLazilyFilteredInstances(src, b)
    return ret
}
```

Once we have that, we can train and test our naive bayes model, as shown in the following code:

```
// Read in the loan training data set into golearn "instances".
trainingData, err := base.ParseCSVToInstances("training.csv", true)
if err != nil {
    log.Fatal(err)
}

// Initialize a new Naive Bayes classifier.
nb := naive.NewBernoulliNBClassifier()

// Fit the Naive Bayes classifier.
nb.Fit(convertToBinary(trainingData))

// Read in the loan test data set into golearn "instances".
// This time we will utilize a template of the previous set
// of instances to validate the format of the test set.
testData, err := base.ParseCSVToTemplatedInstances("test.csv", true,
trainingData)
if err != nil {
    log.Fatal(err)
}

// Make our predictions.
```

```
predictions := nb.Predict(convertToBinary(testData))

// Generate a Confusion Matrix.
cm, err := evaluation.GetConfusionMatrix(testData, predictions)
if err != nil {
    log.Fatal(err)
}

// Retrieve the accuracy.
accuracy := evaluation.GetAccuracy(cm)
fmt.Printf("\nAccuracy: %0.2f\n\n", accuracy)
```

Compiling and running this gives the following accuracy:

```
$ go build
$ ./myprogram

Accuracy: 0.63
```

This is not quite as good as our from-scratch logistic regression. However, there is still some predictive power here. A good exercise would be to add some of the other features from the LendingClub dataset to this model, especially some of the categorical variables. This would likely improve the naive bayes result.

References

General classification:

- `github.com/sjwhitworth/golearn` docs: `https://godoc.org/github.com/sjwhitworth/golearn`

Summary

We have covered a variety of classification models including logistic regression, k-nearest neighbors, decision trees, random forest, and naive bayes. In fact, we even implemented logistic regression from scratch. All of these models have their different strengths and weaknesses, which we have discussed. However, they should provide you with a good set of tools to start doing classification with Go.

In the next chapter, we will discuss yet another type of machine learning called **clustering**. This is the first unsupervised technique that we will discuss, and we will try a few different approaches.

6
Clustering

Often, a set of data can be organized into a set of clusters. For example, you may be able to organize data into clusters that correspond to certain underlying properties (such as demographic properties including age, sex, geography, employment status, and so on) or certain underlying processes (such as browsing, shopping, bot interactions, and other such behaviors on a website). The machine learning techniques to detect and label these clusters are referred to as **clustering** techniques, naturally.

Up to this point, the machine learning algorithms that we have explored have been **supervised**. That is, we have a set of features or attributes paired with a corresponding label or number that we are trying to predict. We use this labeled data to fit our model to the behavior that we already knew about prior to training the model.

Most clustering techniques are **unsupervised**. As opposed to supervised techniques for regression and classification, we often do not know about the clusters in our dataset prior to finding them with a clustering model. Thus, we go into a clustering problem with an unlabeled dataset and an algorithm, and we generate the cluster labels for our data using the clustering algorithm.

Further, clustering techniques are distinguished from other machine learning techniques in that it is rather difficult to say what the *correct* or *accurate* clusters for the given dataset are. Depending on how many clusters you are looking for and the measures that you are using for similarity between data points, you might end up with a variety of sets of clusters, each having some underlying meaning. This does not mean that clustering techniques cannot be evaluated or validated, but it does mean that we need to understand our limitations and be careful when quantifying our results.

Understanding clustering model jargon

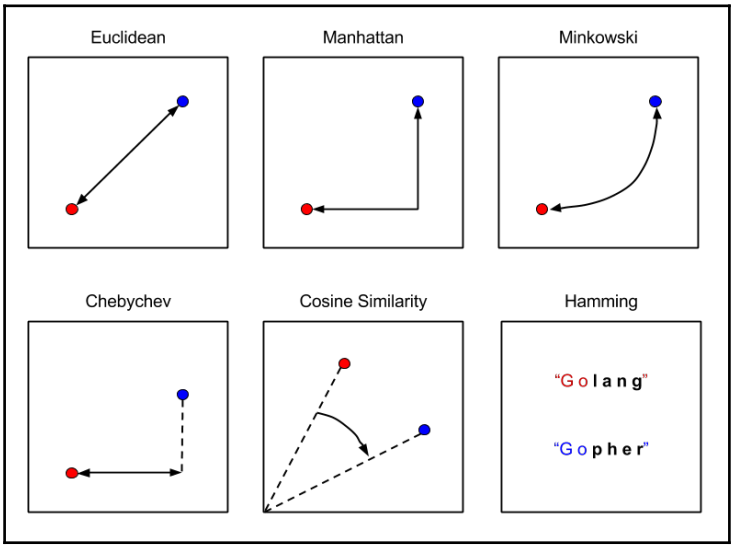
Clustering is quite unique and comes with it's own set of terms, which are shown below. Keep in mind that the following list is only a partial list as there are many different types of clustering with corresponding jargon:

- **Clusters** or **groups**: Each of these clusters or groups is a collection of data points into which our clustering technique organizes our data points.
- **Intra-group** or **intra-cluster**: Clusters resulting from clustering can be evaluated using a measure of similarity between data points and other data points in the same resulting cluster. This is called intra-group or intra-cluster evaluation and similarity.
- **Inter-group**or **inter-cluster**: Clusters resulting from clustering can be evaluated using a measure of dissimilarity between data points and other data points in other resulting clusters. This is called inter-group or inter-cluster evaluation and dissimilarity.
- **Internal criteria**: Often, we do not have a gold standard set of cluster labels that we can use to evaluate our resulting clusters. In these cases, we utilize inter and intra cluster similarities to measure the performance of our clustering technique.
- **External criteria**: In other cases, we might have a gold standard for cluster labels or grouping, such as a standard generated by human judges. These scenarios allow us to evaluate our clustering techniques using the standard, or external, criteria.
- **Distance** or **similarity**: This is a measure of how close two data points are. This could be a Euclidean distance in the space of your features or some other measure of closeness.

Measuring Distance or Similarity

In order to cluster data points together, we need to define and utilize some distance or similarity that quantitatively defines the closeness between data points. Choosing this measure is an essential part of every clustering project because it directly influences how the clusters are generated. Clusters resulting from the use of one similarity measure might be very different from those resulting from the use of another similarity measure.

The most common and simple of these distance measures is the **Euclidean distance** or the **squared Euclidean distance**. This is simply the straight line distance between two data points in your space of features (you might remember this distance as it was also used in our kNN example in `Chapter 5`, *Classification*) or quantity squared. However, there are a whole host of other, sometimes more complicated, distance metrics. A few of these are shown in the following diagram:



For example, the **Manhattan** distance is the absolute *x* distance plus *y* distance between the points, and the **Minkowski** distance generalizes between the Euclidean distance and the Manhattan distance. These distance metrics will be more robust against unusual values (or outliers) in your data, as compared to the Euclidean distance.

Other distance metrics, such as the **Hamming** distance, are applicable to certain kinds of data, such as strings. In the example shown, the Hamming distance between **Golang** and **Gopher** is four because there are four positions in the strings in which the strings are different. Thus, the Hamming distance might be a good choice of distance metric if you are working with text data, such as news articles or tweets.

For our purposes here, we will mostly stick to the Euclidean distance. This distance is implemented in `gonum.org/v1/gonum/floats` via the `Distance()` function. By way of illustration, let's say that we want to calculate the distance between a point at `(1, 2)` and a point at `(3, 4)`. We can do this as follows:

```
// Calculate the Euclidean distance, specified here via
// the last argument in the Distance function.
```

```
distance := floats.Distance([]float64{1, 2}, []float64{3, 4}, 2)

fmt.Printf("\nDistance: %0.2f\n\n", distance)
```

Evaluating clustering techniques

As we are not trying to predict a number or category, our previously discussed evaluation metrics for continuous and discrete variables do not really apply to clustering techniques. That does not mean that we will just avoid measuring the performance of clustering algorithms. We need to know how well our clustering is performing. We just need to introduce a few clustering-specific evaluation metrics.

Internal clustering evaluation

If we do not have a gold standard set of labels for our clusters for comparison, we are stuck with evaluating how well our clustering technique performs using internal criteria. In other words, we can still evaluate our clustering by making similarity and dissimilarity measurements within the clusters themselves.

The first of these internal metrics that we will present here is called the **silhouette coefficient**. The silhouette coefficient can be calculated for each clustered data point as follows:

$$s = \frac{b - a}{\max(a, b)}$$

Here, a is the mean distance between a data point and all other points in the same cluster (the Euclidean distance, for example), and b is the mean distance between a data point and all other points in the cluster nearest to the data point's cluster. The average of this silhouette coefficient for all data points represents how tightly packed the points are in each cluster. This average could be taken per cluster or for data points in all clusters.

Let's try calculating this for the iris dataset, which can be seen as a set of three clusters corresponding to each of the three iris species. First, in order to calculate the silhouette coefficient, we need to know the **centroids** of the three clusters. These centroids are simply the central points of the three clusters (in our four-dimensional feature space), and they will allow us to determine which cluster is the nearest to a certain data point's cluster.

To this end, we need to parse our iris data set file (originally introduced in `Chapter 1`, *Gathering and Organizing Data*), separate our records by cluster label, average the features in each cluster, and then calculate the corresponding centroids. First, we will define a `type` for our centroids:

```
type centroid []float64
```

Then, we can create a map that contains the centroid for each of our iris flowers species using `github.com/kniren/gota/dataframe`:

```
// Pull in the CSV file.
irisFile, err := os.Open("iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Define the names of the three separate species contained in the CSV
file.
speciesNames := []string{
    "Iris-setosa",
    "Iris-versicolor",
    "Iris-virginica",
}

// Create a map to hold our centroid information.
centroids := make(map[string]centroid)

// Filter the dataset into three separate dataframes,
// each corresponding to one of the Iris species.
for _, species := range speciesNames {

    // Filter the original dataset.
    filter := dataframe.F{
        Colname:    "species",
        Comparator: "==",
        Comparando: species,
    }
    filtered := irisDF.Filter(filter)

    // Calculate the mean of features.
    summaryDF := filtered.Describe()

    // Put each dimension's mean into the corresponding centroid.
```

```
        var c centroid
        for _, feature := range summaryDF.Names() {

            // Skip the irrelevant columns.
            if feature == "column" || feature == "species" {
                continue
            }
            c = append(c, summaryDF.Col(feature).Float()[0])
         }

         // Add this centroid to our map.
         centroids[species] = c
    }

    // As a sanity check, output our centroids.
    for _, species := range speciesNames {
        fmt.Printf("%s centroid: %v\n", species, centroids[species])
    }
```

Compiling and running this gives us our centroids:

```
$ go build
$ ./myprogram
Iris-setosa centroid: [5.005999999999999 3.4180000000000006 1.464
0.2439999999999999]
Iris-versicolor centroid: [5.936 2.7700000000000005 4.26
1.3259999999999998]
Iris-virginica centroid: [6.587999999999998 2.9739999999999998 5.552 2.026]
```

Next, we need to actually calculate the silhouette coefficients for each data point. To do this, let's modify the preceding code such that we have access to each filtered set of data points outside of the `for loop`:

```
    // Create a map to hold the filtered dataframe for each cluster.
    clusters := make(map[string]dataframe.DataFrame)

    // Filter the dataset into three separate dataframes,
    // each corresponding to one of the Iris species.
    for _, species := range speciesNames {

        ...

        // Add the filtered dataframe to the map of clusters.
        clusters[species] = filtered

        ...
    }
```

Let's also create a convenience function to retrieve floats values from a row in a `dataframe.DataFrame`:

```
// dfFloatRow retrieves a slice of float values from a DataFrame
// at the given index and for the given column names.
func dfFloatRow(df dataframe.DataFrame, names []string, idx int) []float64
{
        var row []float64
        for _, name := range names {
                row = append(row, df.Col(name).Float()[idx])
        }
        return row
}
```

We can now loop over our records calculating the *a* and *b* that we need for the silhouette coefficients. We will also average the silhouette coefficients to get an overall evaluation metric for our clusters, as shown in the following code:

```
// Convert our labels into a slice of strings and create a slice
// of float column names for convenience.
labels := irisDF.Col("species").Records()
floatColumns := []string{
    "sepal_length",
    "sepal_width",
    "petal_length",
    "petal_width",
}

// Loop over the records accumulating the average silhouette coefficient.
var silhouette float64

for idx, label := range labels {

    // a will store our accumulated value for a.
    var a float64

    // Loop over the data points in the same cluster.
    for i := 0; i < clusters[label].Nrow(); i++ {

        // Get the data point for comparison.
        current := dfFloatRow(irisDF, floatColumns, idx)
        other := dfFloatRow(clusters[label], floatColumns, i)

        // Add to a.
        a += floats.Distance(current, other, 2) /
float64(clusters[label].Nrow())
    }
    // Determine the nearest other cluster.
```

```
        var otherCluster string
        var distanceToCluster float64
        for _, species := range speciesNames {

            // Skip the cluster containing the data point.
            if species == label {
                continue
            }

            // Calculate the distance to the cluster from the current cluster.
            distanceForThisCluster := floats.Distance(centroids[label],
    centroids[species], 2)

            // Replace the current cluster if relevant.
            if distanceToCluster == 0.0 || distanceForThisCluster <
    distanceToCluster {
                otherCluster = species
                distanceToCluster = distanceForThisCluster
            }
        }

        // b will store our accumulated value for b.
        var b float64

        // Loop over the data points in the nearest other cluster.
        for i := 0; i < clusters[otherCluster].Nrow(); i++ {

            // Get the data point for comparison.
            current := dfFloatRow(irisDF, floatColumns, idx)
            other := dfFloatRow(clusters[otherCluster], floatColumns, i)

            // Add to b.
            b += floats.Distance(current, other, 2) /
    float64(clusters[otherCluster].Nrow())
        }

        // Add to the average silhouette coefficient.
        if a > b {
            silhouette += ((b - a) / a) / float64(len(labels))
        }
        silhouette += ((b - a) / b) / float64(len(labels))
    }

    // Output the final average silhouette coeffcient to stdout.
    fmt.Printf("\nAverage Silhouette Coefficient: %0.2f\n\n", silhouette)
```

Compiling and running this example evaluation yields the following:

```
$ go build
$ ./myprogram

Average Silhouette Coefficient: 0.51
```

How do we know if `0.51` is a good or bad average silhouette coefficient? Well, remember that the silhouette coefficient is proportional to the difference between mean intra-cluster distances and mean inter-cluster distances, and it is always going to be somewhere between *0.0* and *1.0*. Thus, higher values (those closer to *1.0*) imply closer packed clusters and are more distinct from other clusters.

Often, we might want to adjust our number of clusters and/or clustering technique to optimize the silhouette score (make it larger, that is). Here, we are working with data that has actually been hand-labeled, so `0.51` has to be a good score for this dataset. For other datasets, it may be higher or lower depending on the existence of clusters in the data and your choice of similarity metric.

The silhouette score is by no means the only way to evaluate our clusters internally. We could actually just use the quantities *a* or *b* from the silhouette score to evaluate the homogeneity of our clusters, or each clusters, dissimilarity with other clusters, respectively. Also, we could use the mean distance between points in a cluster and the centroid of the cluster to measure tightly packed clusters. Still further, we could use a variety of other evaluation metrics that will not be covered here in detail, such as the **Calinski-Harabaz index** (which is discussed further here: `http://datamining.rutgers.edu/publication/internalmeasures.pdf`).

External clustering evaluation

If we have a ground truth or gold standard for our clusters, then we can utilize a variety of external clustering evaluation techniques. This ground truth or gold standard means that we have access to, or can get (via manual human annotation), a set of data points where the true or desired cluster labels have been annotated.

Often, we do not have access to this sort of clustering gold standard, and, thus, we will not cover these sorts of evaluation techniques in detail here. However, if interested or relevant, you can look into the **Adjusted Rand index**, **Mutual Information**, **Fowlkes-Mallows scores**, **completeness**, and **V-measures**, which are all relevant external clustering evaluation metrics (read more details about these here: `https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`).

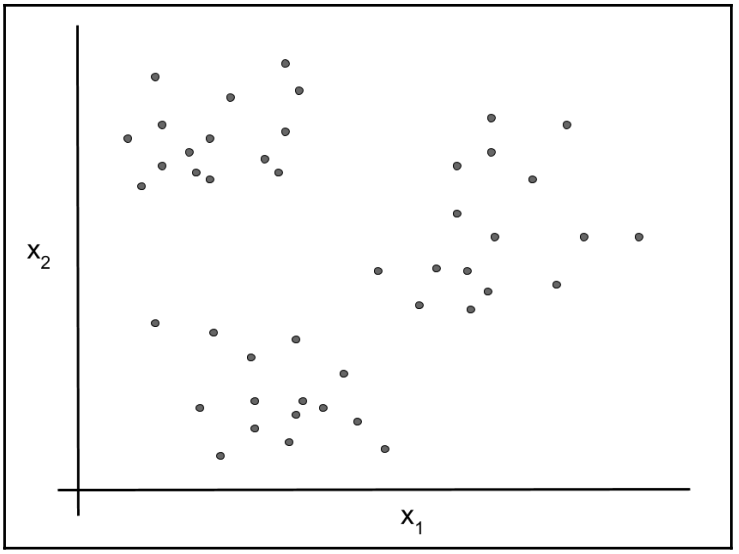
k-means clustering

The first clustering technique that we will cover here, and probably the most well-known clustering technique, is called **k-means** c**lustering**, or just **k-means**. k-means is an iterative method in which data points are clustered around cluster centroids that are adjusted during each iteration. The technique is relatively easy to grasp, but there are some related subtleties that are easy to miss. We will make sure to highlight these as we explore the technique.

As k-means clustering is so easy to implement, there are many proof-of-concept implementations of the algorithm in Go. You can find these by searching for k-means on this link (`https://golanglibs.com/top?q=kmeans`). However, we will utilize a implementation that is recent and fairly straightforward to use, `github.com/mash/gokmeans`.

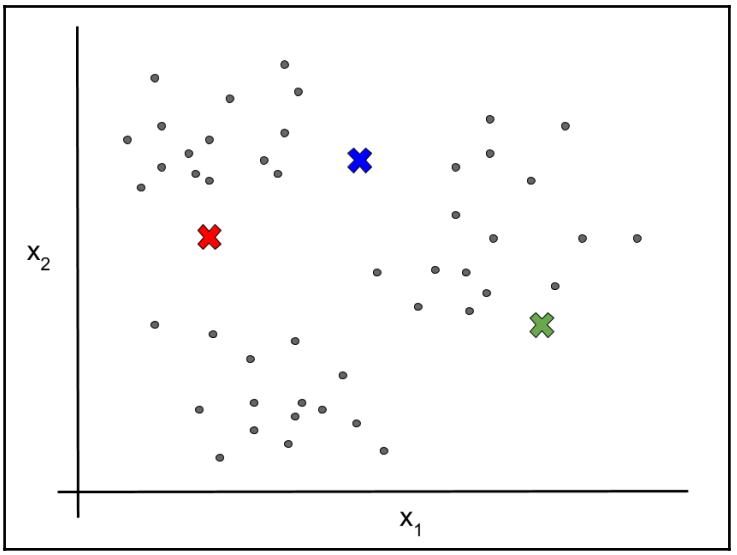
Overview of k-means clustering

Let's say that we have a bunch of data points defined by two variables, x_1 and x_2. These data points naturally exhibit some grouping into clusters, as shown in the following figure:



To automatically cluster these points using k-means, we would first need to choose how many clusters will result from the clustering. This is the parameter *k*, which gives k-means its name. In this case, let's use $k = 3$.

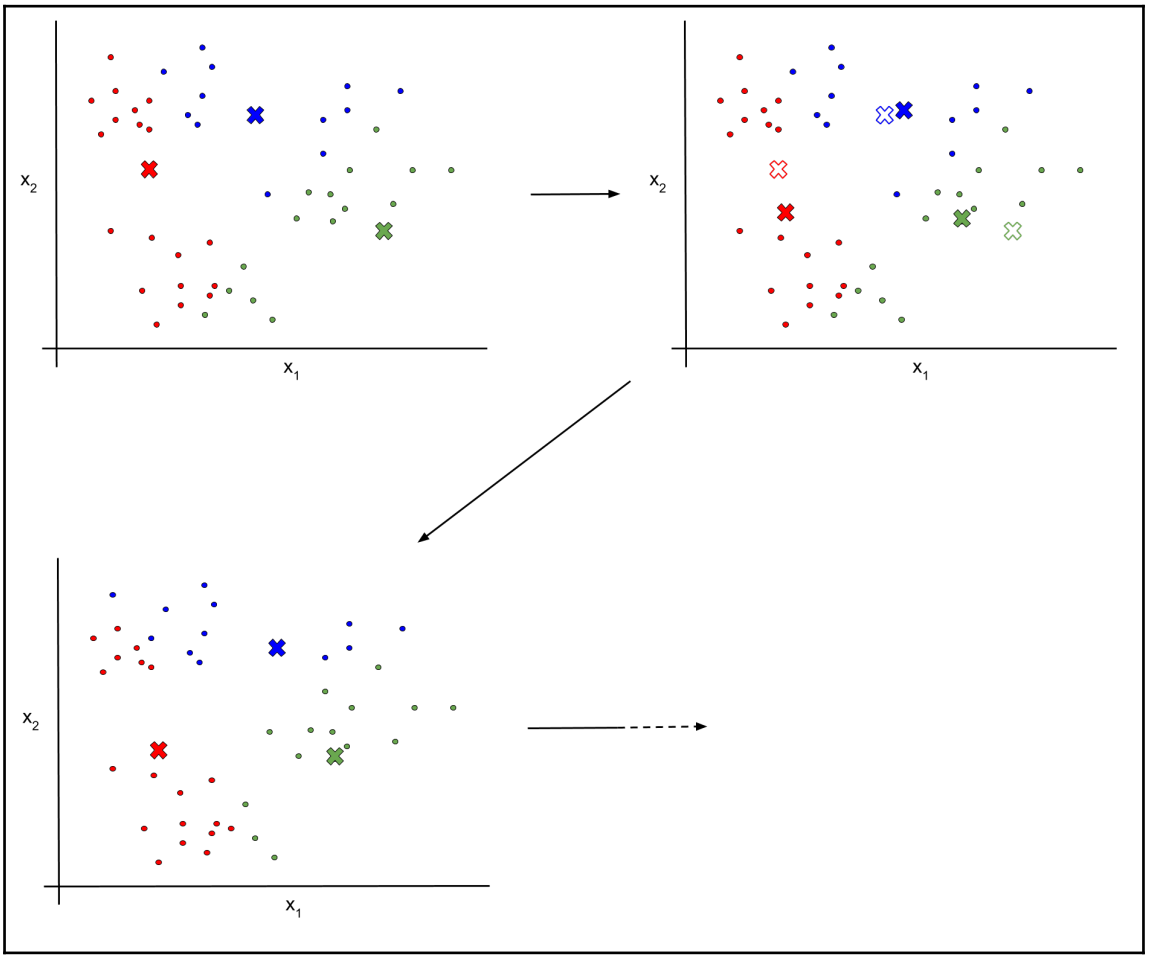
We would then randomly choose the x_1 and x_2 locations of k centroids. These random centroids will serve as our starting point for the algorithm. Such random centroids are shown in the following figure via Xs:



To optimize these centroids and cluster our points, we then iteratively perform the following:

1. Assign each data point to a cluster corresponding to the nearest centroid (as measured by our choice distance metric, such as Euclidean distance).
2. Calculate the mean x_1 and x_2 locations within each cluster.
3. Update each centroid's location to the calculated x_1 and x_2 locations.

Repeat steps one to three until the assignment in step one no longer changes. This process is illustrated in the following figure:

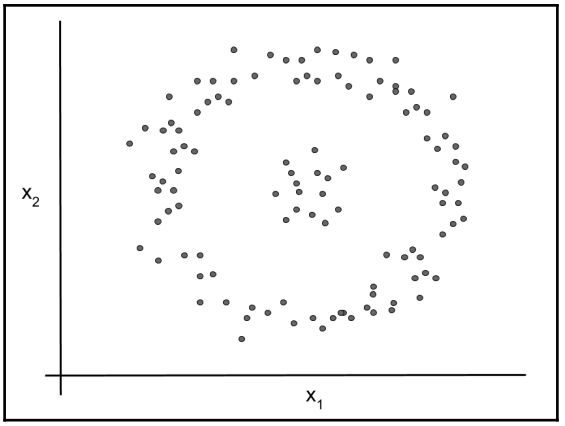


There is only so much that you can illustrate with a static figure, but hopefully this is helpful. If you would like to visually step through the k-means process to better understand the updates, you should check out `http://stanford.edu/class/ee103/visualizations/kmeans/kmeans.html`, which includes an interactive animation of the k-means clustering process.

k-means assumptions and pitfalls

k-means may seem like a very simple algorithm, which it is. However, it does make some underlying assumptions about your data, which are easy to overlook:

- **Spherical** or **spatially grouped clusters**: k-means basically draws spherical or spatially close areas in our feature space to find clusters. This means that for non-spherical clusters (essentially, clusters that do not look like grouped blobs in our features space), k-means is likely to fail. To make this idea more concrete, non-spherical clusters, for which k-means will likely behave poorly, might look like the following:



- **Similar size**: k-means also assumes that your clusters are all of a similar size. Small outlying clusters can lead the simple k-means algorithm going off course to produce strange groupings.

Moreover, there are a couple of pitfalls that we can fall into when using k-means to cluster our data:

- The choice of *k* is up to us. This means that we could choose an illogical *k*, but it also means that we could just continue increasing *k* until we have a cluster for each of our points (which would be pretty good clustering because each point is exactly the same as itself). To help guide your choice of *k*, you should utilize an **elbow graph** approach. In this approach, you increase *k* while calculating your evaluation metric. As you increase *k*, your evaluation metric should keep getting better, but eventually there will be an inflection point that indicates diminishing returns. The ideal *k* is at this elbow.

- It is not guaranteed that k-means will always converge to the same clusters. As you are starting from random centroids, your k-means algorithm could converge to different local minimums on different runs. You should be aware of this and run your k-means algorithm from a variety of initializations to ensure stability.

k-means clustering example

The dataset that we will be using to illustrate clustering techniques is about delivery drivers. The dataset looks like this:

```
$ head fleet_data.csv
Driver_ID,Distance_Feature,Speeding_Feature
3423311935,71.24,28.0
3423313212,52.53,25.0
3423313724,64.54,27.0
3423311373,55.69,22.0
3423310999,54.58,25.0
3423313857,41.91,10.0
3423312432,58.64,20.0
3423311434,52.02,8.0
3423311328,31.25,34.0
```

The first column, `Driver_ID`, includes various anonymous identifications of particular drivers. The second and third columns are attributes that we will utilize in our clusters. The `Distance_Feature` column is a mean distance driven per data, and `Speeding_Feature` is a mean percentage of time during which the driver is driving 5+ miles per hour faster than the speed limit.

The goal of the clustering will be to cluster the delivery drivers into groups based on `Distance_Feature` and `Speeding_Feature`. Remember, this is an unsupervised learning technique, and thus, we do not really know what clusters should or could be formed in the data. The hope is that we will learn something about the drivers that we did not know at the start of the exercise.

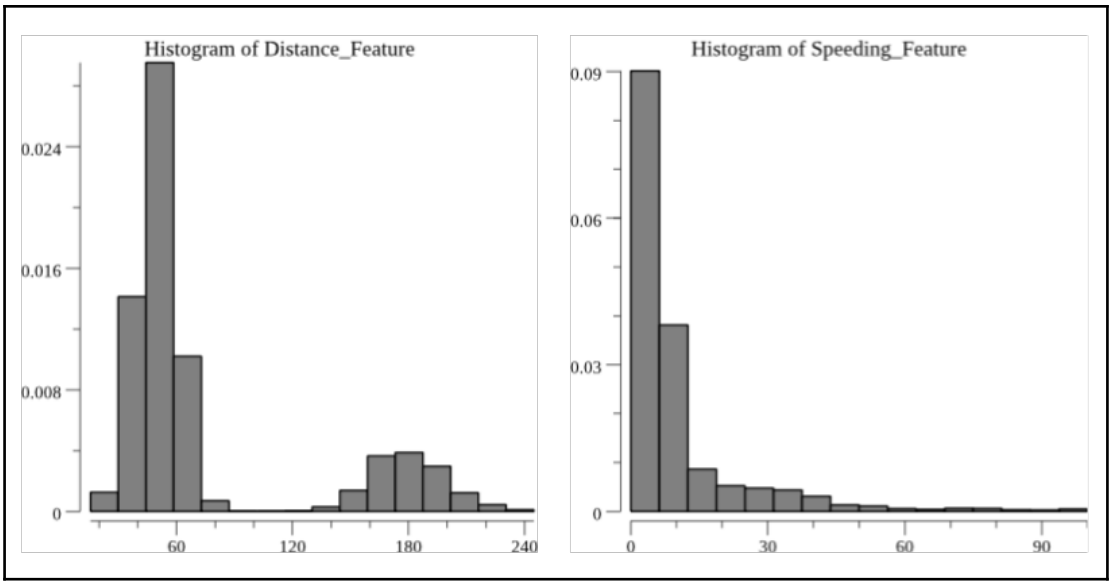
Profiling the data

Yes, you guessed it! We have a new dataset and we need to profile this dataset to learn a little more about it. Let's first calculate summary statistics with `github.com/kniren/dataframe` and create histograms of each feature using `gonum.org/v1/plot`. We have already done this multiple times in `Chapter 4`, *Regression* and `Chapter 5`, *Classification*, so we will not rehash the code here. Let's just look at the results:

```
$ go build
$ ./myprogram
[7x4] DataFrame

    column   Driver_ID          Distance_Feature Speeding_Feature
 0: mean     3423312447.500000 76.041523        10.721000
 1: stddev   1154.844867       53.469563        13.708543
 2: min      3423310448.000000 15.520000        0.000000
 3: 25%      3423311447.000000 45.240000        4.000000
 4: 50%      3423312447.000000 53.330000        6.000000
 5: 75%      3423313447.000000 65.610000        9.000000
 6: max      3423314447.000000 244.790000       100.000000
    <string> <float>           <float>          <float>
```

Wow! Looks like most drivers speed about 10% of the time, which is kind of scary. One driver even appears to speed 100% of the time. I hope I'm not on his route.

The histograms features are shown in the following graph:

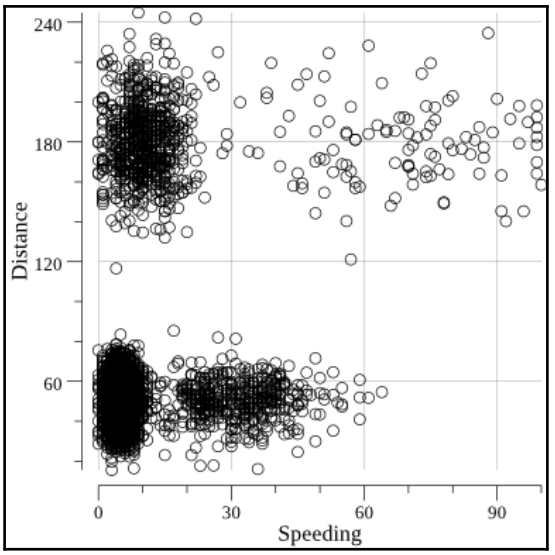


It looks like there is an interesting structure in the `Distance_Feature` data. This will actually factor into our clustering soon, but we can get another view of this structure by creating a scatter plot of our feature space:

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
```

```
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
driverDF := dataframe.ReadCSV(f)

// Extract the distance column.
yVals := driverDF.Col("Distance_Feature").Float()

// pts will hold the values for plotting
pts := make(plotter.XYs, driverDF.Nrow())

// Fill pts with data.
for i, floatVal := range driverDF.Col("Speeding_Feature").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "Speeding"
p.Y.Label.Text = "Distance"
p.Add(plotter.NewGrid())

s, err := plotter.NewScatter(pts)
if err != nil {
    log.Fatal(err)
}
s.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}
s.GlyphStyle.Radius = vg.Points(3)

// Save the plot to a PNG file.
p.Add(s)
if err := p.Save(4*vg.Inch, 4*vg.Inch, "fleet_data_scatter.png"); err !=
nil {
    log.Fatal(err)
}
```

Compiling and running this creates the following scatter plot:



Here, we can see a little bit more of the structure that we saw in the histograms. There appears to be at least two clear clusters of data here. This intuition about our data can serve as a mental check during the formal application of our clustering techniques, and it can give us a starting point to experiment with values of *k*.

Generating clusters with k-means

Now let's get our hands dirty by actually applying k-means clustering to the delivery driver data. To utilize `github.com/mash/gokmeans`, we first need to create a slice of `gokmeans.Node` values, which will be input into the clustering:

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader.
r := csv.NewReader(f)
r.FieldsPerRecord = 3

// Initialize a slice of gokmeans.Node's to
// hold our input data.
var data []gokmeans.Node
```

```
// Loop over the records creating our slice of
// gokmeans.Node's.
for {

    // Read in our record and check for errors.
    record, err := r.Read()
    if err == io.EOF {
        break
    }
    if err != nil {
        log.Fatal(err)
    }

    // Skip the header.
    if record[0] == "Driver_ID" {
        continue
    }

    // Initialize a point.
    var point []float64

    // Fill in our point.
    for i := 1; i < 3; i++ {

        // Parse the float value.
        val, err := strconv.ParseFloat(record[i], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Append this value to our point.
        point = append(point, val)
    }

    // Append our point to the data.
    data = append(data, gokmeans.Node{point[0], point[1]})
}
```

Then, generating our clusters is as easy as calling the `gomeans.Train(...)` function. Specifically, we will call this function with $k = 2$ and a maximum of `50` iterations:

```
// Generate our clusters with k-means.
success, centroids := gokmeans.Train(data, 2, 50)
if !success {
    log.Fatal("Could not generate clusters")
}

// Output the centroids to stdout.
```

```
fmt.Println("The centroids for our clusters are:")
for _, centroid := range centroids {
    fmt.Println(centroid)
}
```

Running all of this together gives the following centroids for the generated clusters:

```
$ go build
$ ./myprogram
The centroids for our clusters are:
[50.04763437499999 8.82875]
[180.01707499999992 18.29]
```

Nice! We have generated our first clusters. Now, we need to move on to evaluating the legitimacy of these clusters.



I have just output the centroids of the clusters here, because that is really all we need to know the group points. If we want to know if a data point is in the first or second cluster, we just need to calculate the distance to those centroids. The closer of the centroids corresponds to the group containing the data point.

Evaluating the generated clusters

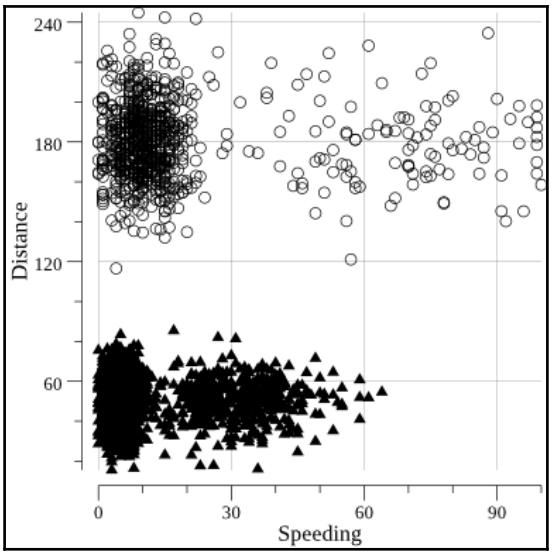
The first way that we can evaluate the clusters that we just generated is visually. Let's create another scatter plot. However, this time let's use different shapes for each of the groups:

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
driverDF := dataframe.ReadCSV(f)

// Extract the distance column.
yVals := driverDF.Col("Distance_Feature").Float()

// clusterOne and clusterTwo will hold the values for plotting.
var clusterOne [][]float64
var clusterTwo [][]float64

// Fill the clusters with data.
for i, xVal := range driverDF.Col("Speeding_Feature").Float() {
```

```
    distanceOne := floats.Distance([]float64{yVals[i], xVal},
[]float64{50.05, 8.83}, 2)
    distanceTwo := floats.Distance([]float64{yVals[i], xVal},
[]float64{180.02, 18.29}, 2)
    if distanceOne < distanceTwo {
        clusterOne = append(clusterOne, []float64{xVal, yVals[i]})
        continue
    }
    clusterTwo = append(clusterTwo, []float64{xVal, yVals[i]})
}

// pts* will hold the values for plotting
ptsOne := make(plotter.XYs, len(clusterOne))
ptsTwo := make(plotter.XYs, len(clusterTwo))

// Fill pts with data.
for i, point := range clusterOne {
    ptsOne[i].X = point[0]
    ptsOne[i].Y = point[1]
}

for i, point := range clusterTwo {
    ptsTwo[i].X = point[0]
    ptsTwo[i].Y = point[1]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "Speeding"
p.Y.Label.Text = "Distance"
p.Add(plotter.NewGrid())

sOne, err := plotter.NewScatter(ptsOne)
if err != nil {
    log.Fatal(err)
}
sOne.GlyphStyle.Radius = vg.Points(3)
sOne.GlyphStyle.Shape = draw.PyramidGlyph{}

sTwo, err := plotter.NewScatter(ptsTwo)
if err != nil {
    log.Fatal(err)
}
sTwo.GlyphStyle.Radius = vg.Points(3)
```

```
// Save the plot to a PNG file.
p.Add(sOne, sTwo)
if err := p.Save(4*vg.Inch, 4*vg.Inch, "fleet_data_clusters.png"); err !=
nil {
    log.Fatal(err)
}
```

This code generates the following scatterplot, which clearly shows our successful clustering:



Qualitatively, we can see that there is one cluster of driver that drive short distances primarily, and one cluster of drivers that drive long distances primarily. These actually correspond to rural and city delivery drivers respectively (or short haul and long haul drivers).

To more quantitatively evaluate our clusters, we can calculate the within-cluster mean distance between points in a cluster and the cluster centroid. To help us in this endeavor, let's create a function that will make things a little easier:

```
// withinClusterMean calculates the mean distance between
// points in a cluster and the centroid of the cluster.
func withinClusterMean(cluster [][]float64, centroid []float64) float64 {

    // meanDistance will hold our result.
    var meanDistance float64

    // Loop over the points in the cluster.
    for _, point := range cluster {
```

```
        meanDistance += floats.Distance(point, centroid, 2) /
float64(len(cluster))
    }

    return meanDistance
}
```

Now, to evaluate our clusters, we simply need to call this function for each cluster:

```
// Output our within cluster metrics.
fmt.Printf("\nCluster 1 Metric: %0.2f\n", withinClusterMean(clusterOne,
[]float64{50.05, 8.83}))
fmt.Printf("\nCluster 2 Metric: %0.2f\n", withinClusterMean(clusterTwo,
[]float64{180.02, 18.29}))
```

Running this gives us the following metrics:

```
$ go build
$ ./myprogram

Cluster 1 Metric: 11.68

Cluster 2 Metric: 23.52
```

As we can see, cluster one (the pink cluster in the scatter plot) is about twice as compact (tightly packed, that is) as cluster two. This is consistent without the plot and gives us a little more quantitative information about the clusters.



Note, that here it was pretty clear that we were looking for two clusters. However, in other cases, the number of clusters may not be clear upfront, especially in cases where you have more features than you can visualize. In these scenarios, it is important that you utilize a method, such as the `elbow` method, to determine a proper *k*. More information about this method can be found at `https://datasciencelab.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/`.

Other clustering techniques

There are a host of other clustering techniques that are not discussed here. These include DBSCAN and Hierarchical clustering. Unfortunately, the current implementations in Go are limited for these other clustering options. DBSCAN is implemented in `https://github.com/sjwhitworth/golearn`, but, to my knowledge, there are no current implementations of other clustering techniques.

This creates a great opportunity for contributions to the community! Clustering techniques are often not complicated and creating an implementation of another clustering technique might be a great way to give back to the Go data science community. Feel free to reach out to the author in Gophers Slack (`@dwhitena`) or other data science gophers in `#data-science` on Gophers Slack if you want to discuss an implementation, ask questions, or get help!

References

Distance metrics and evaluating clusters:

- Evaluation of clustering overview: `https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`
- A comparison of various distance/similarity metrics: `http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0144059`
- Visualizing k-means clustering: `https://www.naftaliharris.com/blog/visualizing-k-means-clustering/`
- `github.com/mash/gokmeans` docs: `https://godoc.org/github.com/mash/gokmeans`

Summary

In this chapter, we have introduced the general principles of clustering, learned how to evaluate generated clusters, and learned how to use a Go implementation of k-means clustering. You should now be in good shape to detect grouping structure in your datasets.

Next, we will discuss the modeling of time series data, such as stock prices, sensor data, and so on.

7

Time Series and Anomaly Detection

Most of the models that we have discussed up to this point predict a property about something based on other properties related to that something. For example, we predicted the species of a flower based on measurements of the flower. We also tried to predict the progression of the disease diabetes in a patient based on medical attributes about that patient.

The premise of time series modeling is different from these types of property prediction problems. Simply put, time series modeling helps us predict the future based on attributes about the past. For example, we may want to predict future stock prices based on previous values of that stock price, or we may want to predict how many users will be on our website at a certain time based on data about how many users were on our website at previous times. This is sometimes called **forecasting**.

The data utilized in time series modeling is typically different from the data utilized in classification, regression, or clustering. A time series model operates on one or more **time series**, as one might expect. This series is a sequential set of properties, attributes, or other numbers paired with their corresponding date and time or a corresponding proxy for a date and time (measurement index or day number, for example). For stock prices, this series would consist of a bunch of (date and time, stock price) pairings.

This time series data is found everywhere in industry and in academia. It is also becoming increasingly important as we explore and develop the **Internet of Things** (**IoT**). Fitness trackers, *smart* devices such as refrigerators, thermostats, cameras, drones, and many other new devices, are producing a staggering amount of time series data.

Of course, you are not restricted to predicting the future with this type of data. There are many other useful things that you can do with time series data including anomaly detection, which will be covered later in this chapter. Anomaly detection attempts to detect unexpected or out-of-the-ordinary events in a time series. These events might correspond to catastrophic weather events, infrastructure failures, viral social media behavior, and so on.

Representing time series data in Go

There are purpose-built systems to store and work with time series data. Some of these are even written in Go, including Prometheus and InfluxDB. However, some of the tooling that we have already utilized in the book is also suitable to handle time series. Specifically, `github.com/kniren/gota/dataframe`, `gonum.org/v1/gonum/floats`, and `gonum.org/v1/gonum/mat` can help us as we are working with time series data.

Take, for example, a dataset that includes a time series representing the number of international air passengers during the years 1949-1960 (available for download at `https://raw.github.com/vincentarelbundock/Rdatasets/master/csv/datasets/AirPassengers.csv`):

```
$ head AirPassengers.csv
time,AirPassengers
1949.0,112
1949.08333333,118
1949.16666667,132
1949.25,129
1949.33333333,121
1949.41666667,135
1949.5,148
1949.58333333,148
1949.66666667,136
```

Here, the `time` column includes a series of times represented by a year along with a decimal, and the `AirPassengers` column includes the number of international air passengers at that `time`. In other words, this is a time series with a pairing of (time, number of passengers).

This is just tabular data and we can represent it perfectly well using a dataframe or matrix. Let's utilize a dataframe for simplicity, as shown in the following code:

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
```

```
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// As a sanity check, display the records to stdout.
// Gota will format the dataframe for pretty printing.
fmt.Println(passengersDF)
```

This will produce the following output:

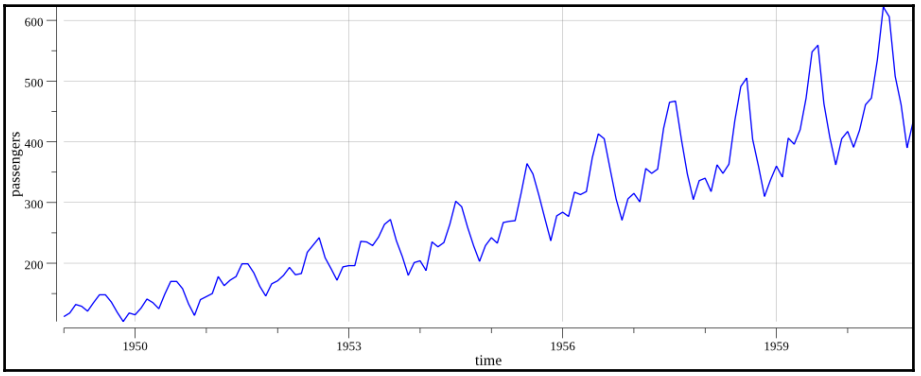
```
$ go build
$ ./myprogram
[144x2] DataFrame

    time        AirPassengers
 0: 1949.000000 112
 1: 1949.083333 118
 2: 1949.166667 132
 3: 1949.250000 129
 4: 1949.333333 121
 5: 1949.416667 135
 6: 1949.500000 148
 7: 1949.583333 148
 8: 1949.666667 136
 9: 1949.750000 119
    ...         ...
    <float>     <int>
```

We could represent the series similarly with `gonum.org/v1/gonum/mat` and, when/if needed, we can convert the dataframe to slices of floats for use with `gonum.org/v1/gonum/floats`. If we wanted to plot the time series, for example, we could convert the columns to floats and produce a plot with `gonum.org/v1/plot`, as shown in the following code:

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Extract the number of passengers column.
yVals := passengersDF.Col("AirPassengers").Float()
```

```
// pts will hold the values for plotting.
pts := make(plotter.XYs, passengersDF.Nrow())

// Fill pts with data.
for i, floatVal := range passengersDF.Col("time").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "passengers"
p.Add(plotter.NewGrid())

// Add the line plot points for the time series.
l, err := plotter.NewLine(pts)
if err != nil {
    log.Fatal(err)
}
l.LineStyle.Width = vg.Points(1)
l.LineStyle.Color = color.RGBA{B: 255, A: 255}

// Save the plot to a PNG file.
p.Add(l)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "passengers_ts.png"); err != nil {
    log.Fatal(err)
}
```

Compiling and running this program produces the following plot of our time series:



As expected, the number of international air passengers increases over time as more and more people begin to travel via airplanes. We can also see that there appear to be bumps or spikes that are repeating over time. We will dive more into these features shortly.

Understanding time series jargon

You are probably noticing by this time in the book that each set of machine learning techniques has an associated set of jargon, and time series is no different.

Here is an explanation of some of this jargon that will be utilized throughout the rest of the chapter:

- **Time**, **datetime**, or **timestamp**: This property is the temporal element of each pairing in our time series. This could be simply a time or it could be a combination of date and time (sometimes referred to as datetime or timestamp). It might also include time zone.
- **Observation**, **measurement**, **signal**, or **random variable**: This is the property that we are trying to forecast and/or otherwise analyze as a function of time.
- **Seasonality**: A time series, such as the time series of air passenger data, may exhibit changes that correspond to seasons (weeks, months, years, and so on). Time series that behave in this manner are said to exhibit some seasonality.
- **Trends**: Time series that gradually increase or decrease over time (separate from seasonal effects) are said to exhibit a trend.
- **Stationary**: A time series that exhibits the same patterns over time, without trends or other gradual changes (such as changes in variance or covariance), is said to be stationary.
- **Time period**: The amount of time between successive observations in the time series, or the difference between one timestamp and the previously occurring timestamp in the series.
- **Auto-regressive model**: This is a model that tries to model a time series process by one or more delayed, or lagged, versions of the same process. For example, an auto-regressive model of stock prices would try to model stock prices by the value of the stock price at previous time intervals.
- **Moving average model**: This is a model that tries to model a times series based on the current and various past values of an imperfectly predictable term, commonly referred to as **error**. For example, this imperfectly predictable term may be some white noise in the time series.

Statistics related to time series

In addition to certain jargon associated with time series, there is an important set of statistics related to time series that we will be relying on as we perform forecasting and anomaly detection. These statistics are mainly related to how values in times series are related to other values in the same time series.

The statistics will help us as we profile our data, which is an important part of any time series modeling project, as it is with all of the other types of modeling that we have covered. Gaining intuition about the behavior of your time series over time, seasonality, and trends is crucial for ensuring that you apply appropriate models and perform mental checks of your results.

Autocorrelation

Autocorrelation is a measure of how correlated a signal is with a delayed version of itself. For example, one or more previous observation of a stock price may be correlated (or change together) with the next observation of the stock price. If this was the case, we would say that the stock price was influenced by itself according to some lag or delay. We could then model the future stock price by a delayed version of itself at the specific lags indicated as highly correlated.

To measure the autocorrelation of a variable x_t with a delayed version of itself (or a version with a lag) x_s, we can utilize the **autocorrelation function** (**ACF**), defined as follows:

$$acf = \frac{(x_{t1} - \overline{x}_t)(x_{s1} - \overline{x}_t) + (x_{t2} - \overline{x}_t)(x_{s2} - \overline{x}_t) + \ldots}{(x_{t1} - \overline{x}_t)^2 + (x_{t2} - \overline{x}_t)^2 + \ldots} \begin{cases} \text{where } x_t \text{ is the series} \\ \textit{where } x_s \textit{ is the time series with a lag or delay, and} \\ \textit{where } \overline{x}_t \textit{ is the mean of } x_t \end{cases}$$

Here, s could represent any lagged version of x. Thus, we can calculate the autocorrelation between x and a version of x that has a lag of a single time period (x_{t-1}), between x and a version of x that has a lag of two time periods (x_{t-2}), and so on. Doing this gives us information about which delayed versions of x are most correlated with x, and thus helps us determine which delayed versions of x might be good candidates for use in modeling future versions of x.

Let's try calculating the first few autocorrelations of our airline passenger time series with itself. To do this, we first need to create a function that will calculate an autocorrelation in our time series for a specific time period lag. Here is an example implementation of this function:

```
// acf calculates the autocorrelation for a series
// at the given lag.
func acf(x []float64, lag int) float64 {

    // Shift the series.
    xAdj := x[lag:len(x)]
    xLag := x[0 : len(x)-lag]

    // numerator will hold our accumulated numerator, and
    // denominator will hold our accumulated denominator.
    var numerator float64
    var denominator float64

    // Calculate the mean of our x values, which will be used
    // in each term of the autocorrelation.
    xBar := stat.Mean(x, nil)

    // Calculate the numerator.
    for idx, xVal := range xAdj {
        numerator += ((xVal - xBar) * (xLag[idx] - xBar))
    }

    // Calculate the denominator.
    for _, xVal := range x {
        denominator += math.Pow(xVal-xBar, 2)
    }

    return numerator / denominator
}
```

We will then loop over a few lags and utilize the `acf()` function to calculate the various autocorrelations. This process is shown in the following code:

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)
```

```
// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("AirPassengers").Float()

// Loop over various values of lag in the series.
fmt.Println("Autocorrelation:")
for i := 1; i < 11; i++ {

    // Shift the series.
    adjusted := passengers[i:len(passengers)]
    lag := passengers[0 : len(passengers)-i]

    // Calculate the autocorrelation.
    ac := stat.Correlation(adjusted, lag, nil)
    fmt.Printf("Lag %d period: %0.2f\n", i, ac)
}
```

This produces the following results:

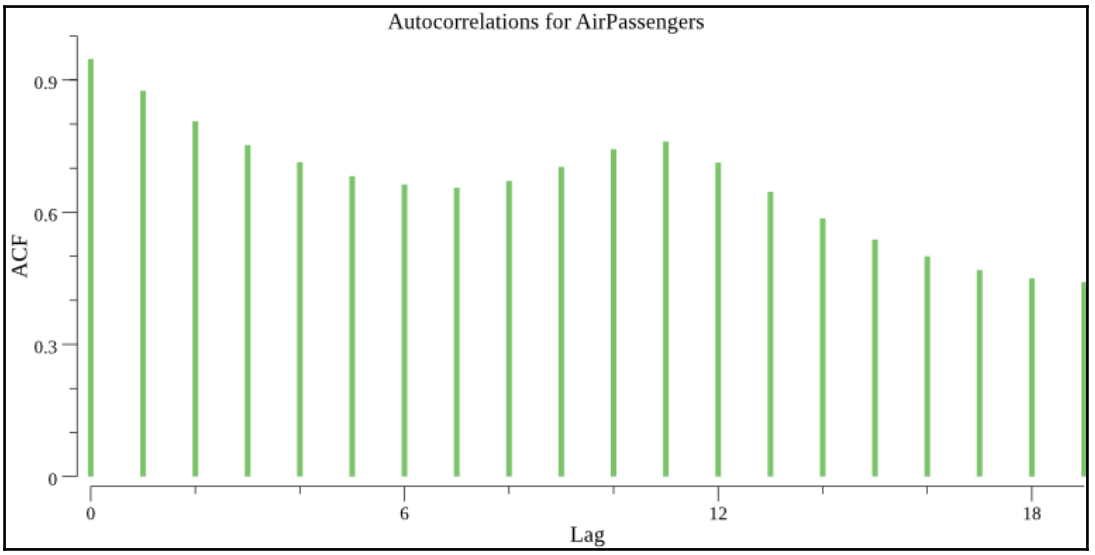
```
$ go build
$ ./myprogram
Autocorrelation:
Lag 1 period: 0.95
Lag 2 period: 0.88
Lag 3 period: 0.81
Lag 4 period: 0.75
Lag 5 period: 0.71
Lag 6 period: 0.68
Lag 7 period: 0.66
Lag 8 period: 0.66
Lag 9 period: 0.67
Lag 10 period: 0.70
```

As we can see, the autocorrelations with lags further back in the series tend to be smaller (although, this is not the case for every lag). However, this information can be a little bit hard to absorb in its numerical form. Let's plot these values as a function of the lag to better visualize the correlations:

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
```

```
passengers := passengersDF.Col("AirPassengers").Float()

// Create a new plot, to plot our autocorrelations.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}

p.Title.Text = "Autocorrelations for AirPassengers"
p.X.Label.Text = "Lag"
p.Y.Label.Text = "ACF"
p.Y.Min = 0
p.Y.Max = 1

w := vg.Points(3)

// Create the points for plotting.
numLags := 20
pts := make(plotter.Values, numLags)

// Loop over various values of lag in the series.
for i := 1; i <= numLags; i++ {

    // Calculate the autocorrelation.
    pts[i-1] = acf(passengers, i)
}

// Add the points to the plot.
bars, err := plotter.NewBarChart(pts, w)
if err != nil {
    log.Fatal(err)
}
bars.LineStyle.Width = vg.Length(0)
bars.Color = plotutil.Color(1)

// Save the plot to a PNG file.
p.Add(bars)
if err := p.Save(8*vg.Inch, 4*vg.Inch, "acf.png"); err != nil {
    log.Fatal(err)
}
```

This code produces the following plot of the ACF:



Notice how the autocorrelations are decreasing generally, but they are staying rather large (well above 0.5) even out to lags of 20 time periods. This is an indication that our time series is not stationary. Indeed, if we look at the previous plot of our time series, it is obviously trending upward. We will deal with this non-stationary behavior later in the chapter, but for now, suffice it to say that the ACF plot is indicating to us that the lagged versions of the number of air passengers are correlated with their non-delayed counterparts.



More generally, the ACF will allow us to determine which type of time series we are modeling. For a process that could be modeled well by an auto-regressive model, we should see that the `acf` function decreases somewhat quickly, but not immediately, as you move to further lags. For a process that could be modeled well by a so-called moving average model, we would see a significant ACF term at the first lag, but then the ACF would die off after that first lag.

For more information on interpreting ACF plots, see `https://coolstatsblog.com/2013/08/07/how-to-use-the-autocorreation-function-acf/`. This post gives some great details, some of which we are not able to cover here.

Partial autocorrelation

As you might expect from the name, partial autocorrelation is related to autocorrelation, but there are some subtle differences. Partial means that this is a conditional sort of correlation. In essence, partial autocorrelation measures the correlation of a series with itself at a certain lag after subtracting off any autocorrelations at intermediate lags. You could think of this as the leftover autocorrelation after intermediate correlations have been removed.

The reason that we might want something like this is that we need more than just the ACF to determine the order of our time series model, assuming that it can be modeled by an auto-regressive model. Let's suppose that, using the ACF, we have determined that we can model our series by an auto-regressive model, because the ACF decays exponentially with the lags. How are we to know if we should model this time series by a version of itself lagged by one time period, or both a version of itself lagged by one time period and two time periods, and so on?

By subtracting out intermediate correlations, we are able to quickly determine any leftover correlation that could be modeled using an auto-regressive model with more terms. If the partial autocorrelation dies off after a first lag, we know that we should be able to model our series based on a single lagged version of itself (lagged at one time period). However, if the partial autocorrelation does not die off after a first lag, we know that we will need to employ multiple lagged versions of the time series in our auto-regressive model.

If we imagine linearly modeling a value in our time series (x_t) by values at successively larger lags in the series (x_{t-1}, x_{t-2}, and so on), our equation would look as follows:

$$x_t = m_1 x_{t-1} + m_2 x_{t-2} + m_3 x_{t-3} + \ldots + b$$

The various coefficients m_1, m_2, and so on, are the partial autocorrelations for a lag of one time period, the partial autocorrelation for the lag at two time periods, and so on, respectively. Thus, all we need to do to calculate the partial autocorrelation for a certain lag is to estimate the linear regression formula that will give us the corresponding coefficient. A function that performs this calculation is called the **partial autocorrelation function** (**PACF**).

Using our favorite linear regression package, `github.com/sajari/regression`, we can create a Go function that implements the PACF as follows:

```
// pacf calculates the partial autocorrelation for a series
// at the given lag.
func pacf(x []float64, lag int) float64 {
```

```
    // Create a regresssion.Regression value needed to train
    // a model using github.com/sajari/regression.
    var r regression.Regression
    r.SetObserved("x")

    // Define the current lag and all of the intermediate lags.
    for i := 0; i < lag; i++ {
        r.SetVar(i, "x"+strconv.Itoa(i))
    }

    // Shift the series.
    xAdj := x[lag:len(x)]

    // Loop over the series creating the data set
    // for the regression.
    for i, xVal := range xAdj {

        // Loop over the intermediate lags to build up
        // our independent variables.
        laggedVariables := make([]float64, lag)
        for idx := 1; idx <= lag; idx++ {

            // Get the lagged series variables.
            laggedVariables[idx-1] = x[lag+i-idx]
        }

        // Add these points to the regression value.
        r.Train(regression.DataPoint(xVal, laggedVariables))
    }

    // Fit the regression.
    r.Run()

    return r.Coeff(lag)
}
```

Then, we can use this `pacf` function to calculate a few values of partial autocorrelation, corresponding to lags for which we previously calculated autocorrelation. This is demonstrated as follows:

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
```

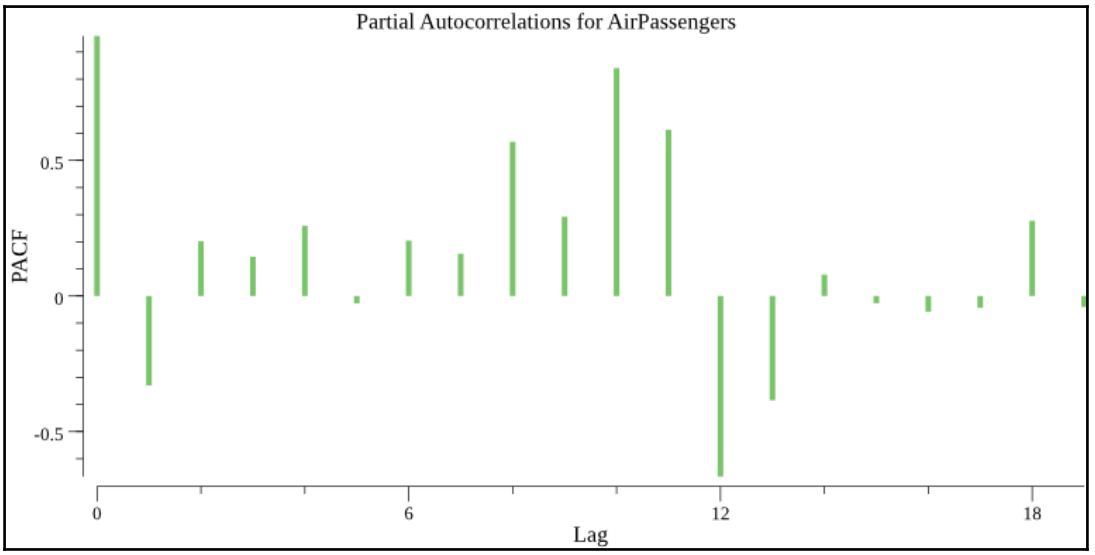
```
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("AirPassengers").Float()

// Loop over various values of lag in the series.
fmt.Println("Partial Autocorrelation:")
for i := 1; i < 11; i++ {

    // Calculate the partial autocorrelation.
    pac := pacf(passengers, i)
    fmt.Printf("Lag %d period: %0.2f\n", i, pac)
}
```

Compiling and running this gives the following values of `Partial Autocorrelation` in our air passengers times series:

```
$ go build
$ ./myprogram
Partial Autocorrelation:
Lag 1 period: 0.96
Lag 2 period: -0.33
Lag 3 period: 0.20
Lag 4 period: 0.15
Lag 5 period: 0.26
Lag 6 period: -0.03
Lag 7 period: 0.20
Lag 8 period: 0.16
Lag 9 period: 0.57
Lag 10 period: 0.29
```

As you can see, the partial autocorrelation dies off quickly after about the second lag. This indicates that there is not much remaining relationship-wise in the time series after factoring in the relationships between the time series and its first and second lags. The partial autocorrelation does not go to *0.0* exactly, but that is expected due to some noise in the data.

To help us better visualize the PACF, let's create another plot. We can do this the exact same way as we did with the ACF, just substituting the `pacf()` function for the `acf()` function. The resulting plot is as follows:



Auto-regressive models for forecasting

The first category of models that we are going to use to try and forecast our time series are called **auto-regressive** (**AR**) models. As already mentioned, we try to model a data point in our time series based on one or more previous points in the series. We are, thus, modeling the time series using the time series itself. This use of the series itself is what distinguishes AR methods from the more general regression methods discussed in `Chapter 4`, *Regression*.

Auto-regressive model overview

You will often see AR models referred to as AR(1), AR(2), and so on. These numbers correspond to the **order** of the AR model or process you are using to model the time series, and it is this order that you can determine by performing autocorrelation and partial autocorrelation analysis.

An AR(1) model attempts to model an observation in your series based on the observation in the same series at a one time period delay:

$$x_t = m_1 x_{t-1} + b$$

An AR(2) model would look as follows:

$$x_t = m_1 x_{t-1} + m_2 x_{t-2} + b$$

An AR(3) model would add another term and so on, all following this pattern.



These formulas might remind you of linear regression, and we will actually use many of the same methods here that we used when creating linear regression models. However, the unique aspects of time series modeling should not be ignored here. It is important to have an intuition of the time-related elements of your data (seasonality, trends, autocorrelations, and so on) and how they influence the AR models.

The order of AR model that you use to model your time series can be determined best by looking at a graph of the PACF. In the graph, you will see the PACF values decay to and then hover around zero. Look at how many lags it takes for the PACF to start hovering around zero, and then utilize an AR order corresponding to that many lags.



Note that some packages to plot the PACF and ACF include horizontal lines indicating the statistical significance of the various lagged terms. I have not included these here, but if you want to quantitatively determine the order for your AR models, you might consider calculating these as further discussed here: `http://www.itl.nist.gov/div898/handbook/eda/section3/autocopl.htm` and `http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4463.htm`.

Auto-regressive model assumptions and pitfalls

The main assumptions of auto-regressive models are as follows:

- **Stationarity**: AR models assume that your time series is stationary. We should not see any trends in the data if we plan on using AR models.
- **Ergodicity**: This fancy term basically means that the statistical properties of the time series, like mean and variance, should not vary or drift over time.

Whatever time series that we are modeling with AR methods should meet these assumptions. However, even when some data (like our air passenger data) that does not meet these assumptions, we can play some differencing tricks to still take advantage of AR models.

Auto-regressive model example

We will try to model our air passenger data using an auto-regressive model. Now, we already know that we are breaking one of the assumptions of AR models in that our data is not stationary. However, we can apply a common trick to make our series stationary, called **differencing**.

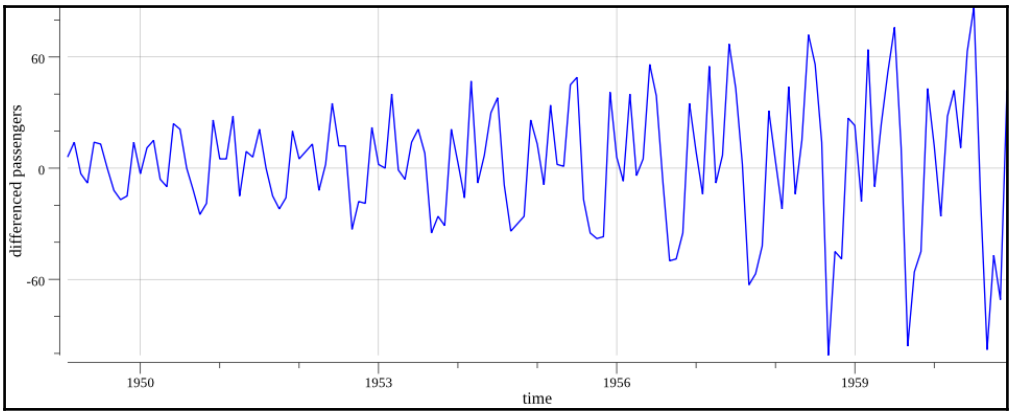
Transforming to a stationary series

To make our time series stationary, we will create a proxy time series where the observation at time period *t* is the observation at time period *t* from our original time series minus the previous observation. Let's difference each observation in this manner and then plot the results to see if it gets rid of the trends in our data. We are also going to output this differenced time series to a new `*.csv` file, as shown in the following code:

```
// as slices of floats.
passengerVals := passengersDF.Col("AirPassengers").Float()
timeVals := passengersDF.Col("time").Float()

// pts will hold the values for plotting.
pts := make(plotter.XYs, passengersDF.Nrow()-1)

// differenced will hold our differenced values
// that will be output to a new CSV file.
var differenced [][]string
differenced = append(differenced, []string{"time",
"differenced_passengers"})

// Fill pts with data.
for i := 1; i < len(passengerVals); i++ {
    pts[i-1].X = timeVals[i]
    pts[i-1].Y = passengerVals[i] - passengerVals[i-1]
    differenced = append(differenced, []string{
        strconv.FormatFloat(timeVals[i], 'f', -1, 64),
        strconv.FormatFloat(passengerVals[i]-passengerVals[i-1], 'f', -1,
64),
    })
}
```

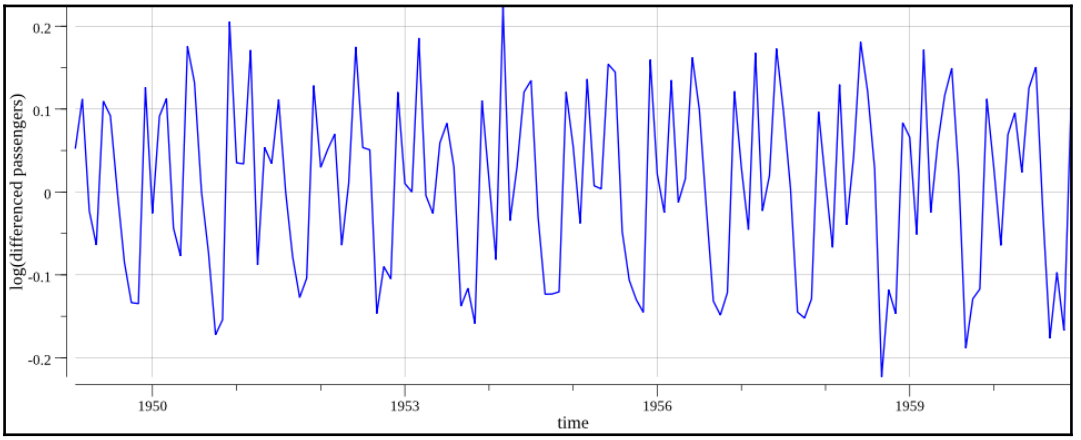
```
// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "differenced passengers"
p.Add(plotter.NewGrid())

// Add the line plot points for the time series.
l, err := plotter.NewLine(pts)
if err != nil {
    log.Fatal(err)
}
l.LineStyle.Width = vg.Points(1)
l.LineStyle.Color = color.RGBA{B: 255, A: 255}

// Save the plot to a PNG file.
p.Add(l)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "diff_passengers_ts.png"); err !=
nil {
    log.Fatal(err)
}

// Save the differenced data out to a new CSV.
f, err := os.Create("diff_series.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

w := csv.NewWriter(f)
w.WriteAll(differenced)

if err := w.Error(); err != nil {
    log.Fatal(err)
}
```

This results in the following plot:



Here, we can see that we have basically removed all signs of the upward trend that was in the original time series. However, there still appears to be a problem related to variance. The differenced time series appears to have an increasing variance about the mean as time gets larger, which breaks our ergodicity assumption.

To deal with the increase in variance, we can further transform our time series using a log or power transformation that penalizes the larger values later in the time series. Let's add this log transform, replot the log of the differenced series, and then save the resulting data to a file called `log_diff_series.csv`. The code to accomplish this is the same as the previous code snippet, except we use `math.Log()` to transform each value, so we will spare the details. The following is the resulting plot:



Awesome! Now we have what looks to be a stationary series that we can utilize in our AR models. Note, we are looking at this qualitatively for the most part in this example, but there are quantitative tests for stationarity (the Dickey–Fuller test, for example).

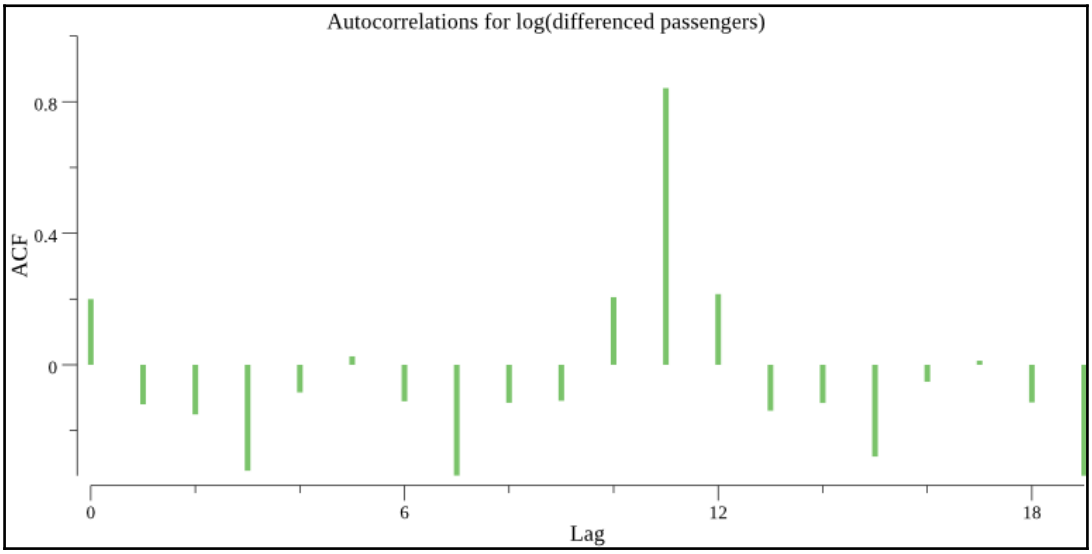


We have transformed our data here with both a difference and a log transformation. This allowed us to fit within the assumptions of the AR model, but it also made our data and our eventual model a little less interpretable. It's harder to think about the log of a differenced time series than the time series itself. We had a justification for this trade-off here, but the trade-off should be noted, and the hope is to avoid such obfuscations where we can.

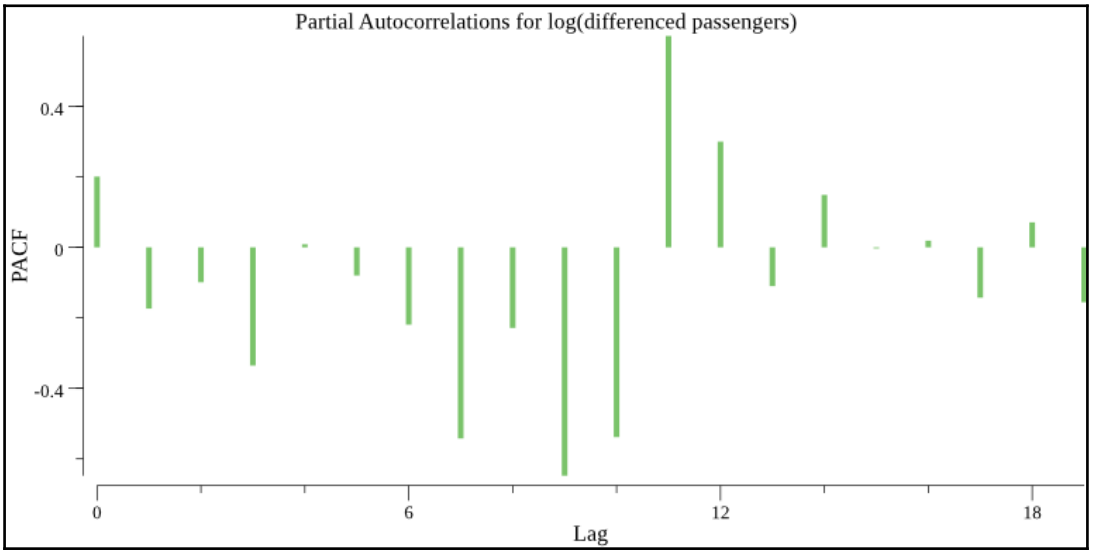
Analyzing the ACF and choosing an AR order

Now that we have a stationary series that fits within the assumptions of our model, let's revisit our ACF and PACF plots to see what has changed. We can utilize the same code that we used to plot the ACF and PACF previously, but this time we will use our transformed series.

Here are the resulting ACF plot:



Here are the resulting PACF plot:



The first thing we notice is that the ACF plot no longer has a slow decay from *1.0* as the lags get larger and larger. The ACF plot decays and fluctuates around *0.0*. We will come back to the ACF in the next section.

Next, we can see that the PACF also decays down to *0.0* and fluctuates around 0.0 thereafter. To choose the order of our AR model, we want to examine where the PACF plot appears to cross the zero line for the first time. In our case, this appears to be after the second lag period, and thus, we might want to consider using an AR(2) model to model this time series auto-regressively.

Fitting and evaluating an AR(2) model

We have already seen that PACF gives us the coefficients for various orders in our AR model. Taking advantage of this, we can get the coefficients for our first and second lag terms along with the intercept (or error term) in our model using a slightly modified version of the `pacf()` function shown in the following code:

```
// autoregressive calculates an AR model for a series
// at a given order.
func autoregressive(x []float64, lag int) ([]float64, float64) {

    // Create a regresssion.Regression value needed to train
    // a model using github.com/sajari/regression.
```

```
    var r regression.Regression
    r.SetObserved("x")

    // Define the current lag and all of the intermediate lags.
    for i := 0; i < lag; i++ {
        r.SetVar(i, "x"+strconv.Itoa(i))
    }

    // Shift the series.
    xAdj := x[lag:len(x)]

    // Loop over the series creating the data set
    // for the regression.
    for i, xVal := range xAdj {

        // Loop over the intermediate lags to build up
        // our independent variables.
        laggedVariables := make([]float64, lag)
        for idx := 1; idx <= lag; idx++ {

            // Get the lagged series variables.
            laggedVariables[idx-1] = x[lag+i-idx]
         }

         // Add these points to the regression value.
         r.Train(regression.DataPoint(xVal, laggedVariables))
    }

    // Fit the regression.
    r.Run()

    // coeff hold the coefficients for our lags.
    var coeff []float64
    for i := 1; i <= lag; i++ {
        coeff = append(coeff, r.Coeff(i))
    }

    return coeff, r.Coeff(0)
}
```

We can then call this on our log differenced series to get our trained AR(2) model coefficients:

```
// Open the CSV file.
passengersFile, err := os.Open("log_diff_series.csv")
if err != nil {
    log.Fatal(err)
}
```

```
defer passengersFile.Close()
// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("log_differenced_passengers").Float()

// Calculate the coefficients for lag 1 and 2 and
// our error.
coeffs, intercept := autoregressive(passengers, 2)

// Output the AR(2) model to stdout.
fmt.Printf("\nlog(x(t)) - log(x(t-1)) = %0.6f + lag1*%0.6f +
lag2*%0.6f\n\n", intercept, coeffs[0], coeffs[1])
```

Compiling and running this training gives us the following AR(2) formula for the log of the differenced passenger counts:

```
$ go build
$ ./myprogram

log(x(t)) - log(x(t-1)) = 0.008159 + lag1*0.234953 + lag2*-0.173682
```

To evaluate this AR(2) model, we can calculate the **Mean Absolute Error** (**MAE**), similar to how we calculated it for linear regression models. Specifically, we will compute predicted passenger count values paired with our observed passenger count values, and then we will calculate the error and accumulate the MAE.

First, let's calculate our transformed (log and differenced) predictions:

```
// Open the log differenced dataset file.
transFile, err := os.Open("log_diff_series.csv")
if err != nil {
    log.Fatal(err)
}
defer transFile.Close()

// Create a CSV reader reading from the opened file.
transReader := csv.NewReader(transFile)

// Read in all of the CSV records
transReader.FieldsPerRecord = 2
transData, err := transReader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the data predicting the transformed
```

```
// observations.
var transPredictions []float64
for i, _ := range transData {

    // Skip the header and the first two observations
    // (because we need two lags to make a prediction).
    if i == 0 || i == 1 || i == 2 {
        continue
    }

    // Parse the first lag.
    lagOne, err := strconv.ParseFloat(transData[i-1][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the second lag.
    lagTwo, err := strconv.ParseFloat(transData[i-2][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Predict the transformed variable with our trained AR model.
    transPredictions = append(transPredictions,
0.008159+0.234953*lagOne-0.173682*lagTwo)
}
```

Now, in order to calculate our MAE, we need to transform these predictions back to normal passenger counts (such that we can compare them directly to our original time series). The reverse transform of our log and differenced data involves calculating cumulative sums in the transformed series, adding them back to the base series values, and then taking an exponential. This reverse transform, the accumulation of the MAE, and the aggregation of points to plot our observations and predictions is as follows:

```
// Open the original dataset file.
origFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer origFile.Close()

// Create a CSV reader reading from the opened file.
origReader := csv.NewReader(origFile)

// Read in all of the CSV records
origReader.FieldsPerRecord = 2
origData, err := origReader.ReadAll()
```

```
if err != nil {
    log.Fatal(err)
}

// pts* will hold the values for plotting.
ptsObs := make(plotter.XYs, len(transPredictions))
ptsPred := make(plotter.XYs, len(transPredictions))

// Reverse the transformation and calculate the MAE.
var mAE float64
var cumSum float64
for i := 4; i <= len(origData)-1; i++ {

    // Parse the original observation.
    observed, err := strconv.ParseFloat(origData[i][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the original date.
    date, err := strconv.ParseFloat(origData[i][0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Get the cumulative sum up to the index in
    // the transformed predictions.
    cumSum += transPredictions[i-4]

    // Calculate the reverse transformed prediction.
    predicted := math.Exp(math.Log(observed) + cumSum)

    // Accumulate the MAE.
    mAE += math.Abs(observed-predicted) / float64(len(transPredictions))

    // Fill in the points for plotting.
    ptsObs[i-4].X = date
    ptsPred[i-4].X = date
    ptsObs[i-4].Y = observed
    ptsPred[i-4].Y = predicted
}
```

Then, let's output the MAE to stand out and save a line plot of the observed and predicted values:

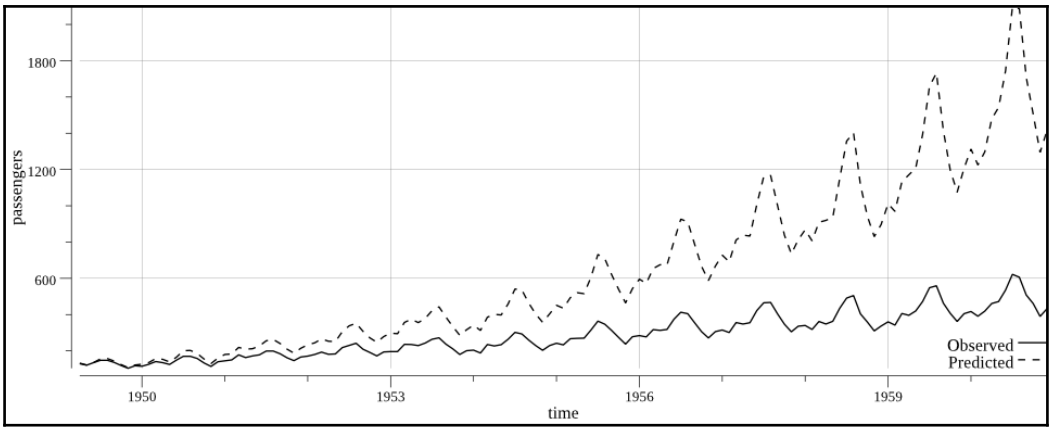
```
// Output the MAE to standard out.
fmt.Printf("\nMAE = %0.2f\n\n", mAE)
```

```
// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "passengers"
p.Add(plotter.NewGrid())

// Add the line plot points for the time series.
lObs, err := plotter.NewLine(ptsObs)
if err != nil {
    log.Fatal(err)
}
lObs.LineStyle.Width = vg.Points(1)

lPred, err := plotter.NewLine(ptsPred)
if err != nil {
    log.Fatal(err)
}
lPred.LineStyle.Width = vg.Points(1)
lPred.LineStyle.Dashes = []vg.Length{vg.Points(5), vg.Points(5)}

// Save the plot to a PNG file.
p.Add(lObs, lPred)
p.Legend.Add("Observed", lObs)
p.Legend.Add("Predicted", lPred)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "passengers_ts.png"); err != nil {
    log.Fatal(err)
}
```

Compiling this and running it results in the following `MAE`:

```
$ go build
$ ./myprogram

MAE = 355.20
```

If you remember from our original visualization of this series, the passenger counts range from just above zero to just above 600. Thus, an MAE of approximately 355 is not super great. To get a more complete view of how our predictions and observations line up, however, let's look at the plot generated by the preceding code:



As you can see, our model overpredicts the number of air passengers, especially as time goes on. The model does exhibit some of the structure that is seen in the original data and it produces a similar trend. Yet, it appears that we may need a slightly more sophisticated model to more realistically represent this series.



No model is perfect and we have tried a relatively simple time series model here. It's good that we tried to stick with a simple and interpretable model, but our evaluation results would probably motivate us to refactor our model in a real-world scenario. Refactoring is good! It means that we learned something.

Auto-regressive moving averages and other time series models

The model that we tried earlier was a relatively simple pure auto-regressive model. However, we are not stuck with using auto-regression or pure auto-regression alone in our time series models. As with other classes of machine learning models covered in this book, there is a whole zoo of time series techniques, and we cannot cover them all here. However, we did want to mention a few notable techniques that you could explore as you follow up on this material.

Auto-regressive models are often combined with models called **moving average models**. When these are combined, they are often referred to as **auto-regressive moving average** (**ARMA**) or **auto-regressive integrated moving average** (**ARIMA**) models. The moving average part of ARMA/ARIMA models allows you to capture the effects of things like white noise or other error terms in your time series, which would actually improve our AR(2) model for air passengers.

Unfortunately, at the time of writing this content, no out of the box package exists to perform ARIMA in Go. As mentioned earlier, the auto-regressive part is relatively easy, but the moving average fitting is slightly more complicated. This is another great place to jump in with contributions!

There are also time series models that are outside of the realm of ARIMA models. For example, the **Holt-Winters method** attempts to capture seasonality in time series data via a forecast equation and three smoothing equations. There are preliminary implementations of the Holt-Winters method in `github.com/datastream/holtwinters` and `github.com/dgryski/go-holtwinters`. These likely need to be further maintained and productionized, but they serve as a starting point.

Anomaly detection

As mentioned in the introduction to this chapter, we might not always be interested in forecasting a time series. We might want to detect anomalous behavior in a time series. For example, we might want to know when out of the ordinary bursts of traffic come across our network, or we may want an alert when out of the ordinary numbers of users are attempting certain things inside of our application. These events could be tied to security concerns or may just be used to adjust our infrastructure or application settings.

Thankfully, due to Go's history of usage in monitoring and infrastructure, there are a variety of Go-based options to detect anomalies in time series data. This tooling has been used in production to detect anomalous behavior while monitoring infrastructure and applications and, although there are more tools than can be mentioned here, I will highlight a couple.

First, the InfluxDB (`https://www.influxdata.com/`) and Prometheus (`https://prometheus.io/`) ecosystems have a variety of options for anomaly detection. Both InfluxDB and Prometheus offer open source, Go-based time series databases and related tooling. They are useful to monitor infrastructure and applications, and they have widespread use both in the Go community and outside of the Go community. For example, if you are interested in using InfluxDB, you can use `github.com/nathanielc/morgoth` for anomaly detection.

This package implements the **Lossy Counting Algorithm** (**LCA**). On the Prometheus side, you could utilize a query-based approach as further discussed on `https://prometheus.io/blog/2015/06/18/practical-anomaly-detection/`.

There are a variety of standalone Go packages for anomaly detection as well, including `github.com/lytics/anomalyzer` and `github.com/sec51/goanomaly`. More specifically, `github.com/lytics/anomalyzer` implements a variety of tests to determine if an observation in your series is anomalous, including tests based on the cumulative distribution functions, bootstrap permutations, permuted rank-sums, relative magnitudes, and more.

To detect anomalies with `github.com/lytics/anomalyzer`, we need to create some configurations and an `anomalyzer.Anomalyzer` value. Once we have done this, detecting an anomaly is as simple as calling the `Push()` method on the `anomalyzer.Anomalyzer` value, as shown in the following code:

```
// Initialize an AnomalyzerConf value with
// configurations such as which anomaly detection
// methods we want to use.
conf := &anomalyzer.AnomalyzerConf{
    Sensitivity: 0.1,
    UpperBound:  5,
    LowerBound:  anomalyzer.NA, // ignore the lower bound
    ActiveSize:  1,
    NSeasons:    4,
    Methods:     []string{"diff", "fence", "highrank", "lowrank",
"magnitude"},
}

// Create a time series of periodic observations
// as a slice of floats.  This could come from a
// database or file, as utilized in earlier examples.
ts := []float64{0.1, 0.2, 0.5, 0.12, 0.38, 0.9, 0.74}

// Create a new anomalyzer based on the existing
// time series values and configuration.
anom, err := anomalyzer.NewAnomalyzer(conf, ts)
if err != nil {
    log.Fatal(err)
}

// Supply a new observed value to the Anomalyzer.
// The Anomalyzer will analyze the value in reference
// to pre-existing values in the series and output
// a probability of the value being anomalous.
prob := anom.Push(15.2)
```

```
fmt.Printf("Probability of 15.2 being anomalous: %0.2f\n", prob)

prob = anom.Push(0.43)
fmt.Printf("Probability of 0.33 being anomalous: %0.2f\n", prob)
```

Compiling and running this anomaly detection yields the following:

```
$ go build
$ ./myprogram
Probability of 15.2 being anomalous: 0.98
Probability of 0.33 being anomalous: 0.80
```

References

Time series statistics (ACF and PACF):

- How to use ACF: `https://coolstatsblog.com/2013/08/07/how-to-use-the-autocorreation-function-acf/`
- Identifying the number of AR or MA terms in an ARIMA model: `https://people.duke.edu/~rnau/411arim3.htm`

Auto-regressive models:

- A more mathematical introduction to AR models: `https://onlinecourses.science.psu.edu/stat501/node/358`
- `github.com/sajari/regression` docs: `https://godoc.org/github.com/sajari/regression`

ARMA/ARIMA models:

- Introduction to ARIMA: `https://people.duke.edu/~rnau/411arim.htm`

Anomaly detection:

- InfluxDB: `https://www.influxdata.com/`
- Prometheus: `https://prometheus.io/`
- `github.com/lytics/anomalyzer` docs: `https://godoc.org/github.com/lytics/anomalyzer`
- `github.com/sec51/goanomaly` docs: `https://godoc.org/github.com/sec51/goanomaly`

Summary

Well, that was timely! We now know what time series data is, how to represent it in Go, how to make some forecasts, and how to detect anomalies in our time series data. These skills will come in useful anytime you are working with data that is changing with time, whether its data related to stock prices, or monitoring data related to your infrastructure.

In the next chapter, we will level up our Go-based machine learning by looking at a few advanced techniques, including neural networks and deep learning.

8

Neural Networks and Deep Learning

In this book, we have talked a lot about training, or teaching, machines to make predictions. To this end, we have employed a variety of useful and interesting algorithms including various types of regression, decisions trees, and nearest neighbors. However, let's take a step back and think about what entity we might want to mimic if we are trying to make accurate predictions and learn about data.

Well, the most obvious answer to this question is that we should mimic our own brains. We as humans have a natural ability to recognize objects, predict quantities, recognize frauds, and more, and these are all things that we would like machines to do artificially. Granted, we are not perfect at these activities, but we are pretty good!

This type of thinking was what lead to the development of **artificial neural networks** (also known as **neural networks** or just **neural nets**). These models attempt to roughly mimic certain structures in our brains, such as **neurons**. They have been widely successful across industries and are currently being applied to solve a variety of interesting problems.

Recently, more specialized and complicated types of neural networks have been attracting a lot of interest and attention. These neural networks fall under the category of **deep learning** and generally have a **deeper** structure than what would be considered regular neural networks. That is, they have many hidden layers of structure and could be parameterized with tens of millions of parameters or weights.

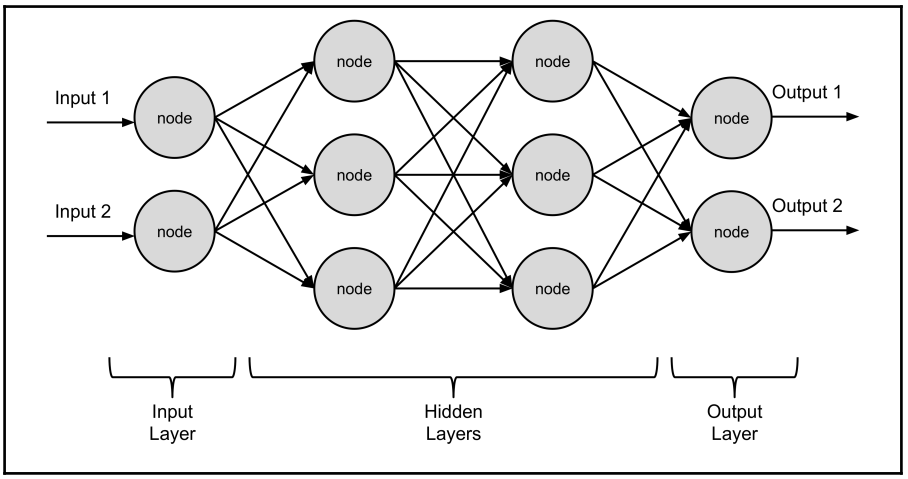
We will attempt to introduce both Go-based neural networks and deep learning models in this chapter. These topics are incredibly broad and there are entire books on deep learning alone. Thus, we will only be touching the surface here. That being said, this following content should provide you with a solid starting point to build neural networks in Go.

Understanding neural net jargon

There are a huge variety of neural network flavors, and each of these flavors has its own set of jargon. However, there is some common jargon that we should know regardless of the type of neural network that we are utilizing. This jargon is presented in the following points:

- **Nodes**, **perceptrons**, or **neurons**: These interchangeable terms refer to the basic building blocks of a neural network. Each node or neuron takes in input data and performs an operation on this data. After performing the operation, the node/neuron may or may not pass the results of the operation on to other nodes/neurons.
- **Activation**: The output or values associated with the operation of a node.
- **Activation function**: The definition of the function that transforms the inputs to a node into the output, or activation.
- **Weights** or **biases**: These values define the relationships between input and output data in the activation function.
- **Input layer**: The input layer of a neural network includes a series of nodes that take the initial input into the neural network model (a series of features or attributes, for example).
- **Output layer**: The output layer of a neural network includes a series of nodes that take information passed inside the neural network and transform it into a final output.
- **Hidden layers**: These layers of a neural network exist between the input and output layers and are thus **hidden** from outside inputs or outputs.
- **Feedforward** or **feed forward**: This refers to a scenario in which data is fed into the input layer of a neural network and transferred in a forward direction to the output layer (without any cycles).
- **Backpropagation:** This is a method for training neural network models that involves feeding forward values through the network, calculating generated errors, and then transferring changes based on these errors back into the network.
- **Architecture**: The overall structure of how neurons are connected together in the neural network is called an **architecture**.

To solidify these concepts, consider the following schematic of a neural network:



This is a basic feedforward (that is, acyclic or recursive) neural network. It has two hidden layers, accepts two inputs, and outputs two class values (or results).

Don't worry if all this jargon seems a little bit overwhelming right now. We will be looking at a concrete example next that should solidify all of these pieces. Also, if you get down in the weeds with the various examples in this chapter and get confused with terminology, circle back here as a reminder.

Building a simple neural network

Many neural network packages and applications of neural networks treat the models as **black boxes**. That is, people tend to utilize some framework that allows them to quickly build a neural network using a bunch of default values and automation. They are often able to produce some results, but this sort of convenience usually does not build much intuition about how the models actually work. As a result, when the models do not behave as expected, it's very hard to understand why they might be making weird predictions or having trouble converging.

Before jumping into more complicated neural networks, let's build up some basic intuition about neural networks such that we do not fall into this pattern. We are going to build a simple neural network from scratch to learn about the basic components of neural networks and how they operate together.



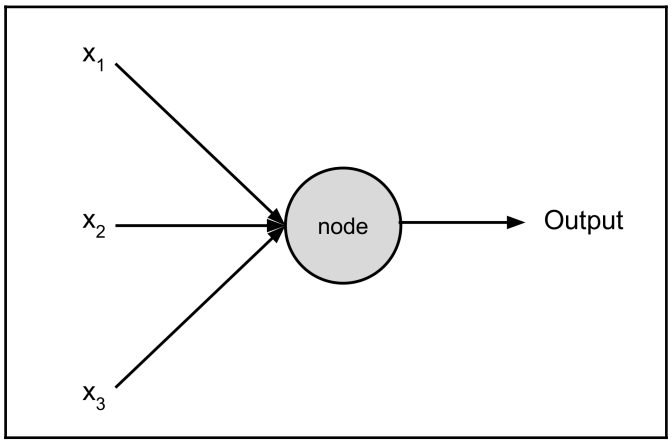
Note: that even though we are going to build our neural net from scratch here (which you may also want to do in certain scenarios), there are a variety of Go packages that help you build, train, and make predictions with neural networks. These include `github.com/tleyden/neurgo`, `github.com/fxsjy/gonn`, `github.com/NOX73/go-neural`, `github.com/milosgajdos83/gosom`, `github.com/made2591/go-perceptron-go`, and `github.com/chewxy/gorgonia`.



We are going to utilize neural networks for classification in this chapter. However, neural nets can also be utilized to perform regression. You can learn more about that topic here: `https://heuristically.wordpress.com/2011/11/17/using-neural-network-for-regression/`.

Nodes in the network

The nodes, or neurons, of our neural network have a relatively simple functionality in and of themselves. Each neuron will take in one or more values (x_1, x_2, and so on), combine these values according to an activation function, and produce a single output. The following is the output pictured:



How exactly should we combine the inputs to get the output? Well, we need a method to combine the inputs that is adjustable (such that we can train the model), and we have already seen that combining variables using coefficients and an intercept is one trainable way to combine inputs. Just think back to `Chapter 4`, *Regression*. In this spirit, we are going to combine the inputs linearly with some coefficients (**weights**) and an intercept (the **bias**):

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Here, w_1, w_2, and so on are our weights and b is the bias.

This combination of inputs is a good start, but it is linear at the end of the day, and thus not able to model non-linear relationships between the input and output. To introduce some non-linearity, we are going to apply an activation function to this linear combination of inputs. The activation function that we will use here is similar to the logistic function that was introduced in `Chapter 5`, *Classification*. In the context of neural networks and in the following form, the logistic function is referred to as the **sigmoid function**:

$$sigmoid\left(x\right) = \frac{1}{1 + e^{-x}}$$

The `sigmoid` function is a good choice for use in our node because it introduces non-linearity, but it also has a limited range (between *0* and *1*), has a simply defined derivative (which we will use during training of the network), and it can have a probabilistic interpretation (as discussed further in `Chapter 5`, *Classification*).



The `sigmoid` function is by no means the only choice for an activation function in a neural network. Other popular choices include the hyperbolic tangent function, softmax, and rectified linear units. Choices of activation functions along with their advantages and disadvantages are further discussed at `https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f`.

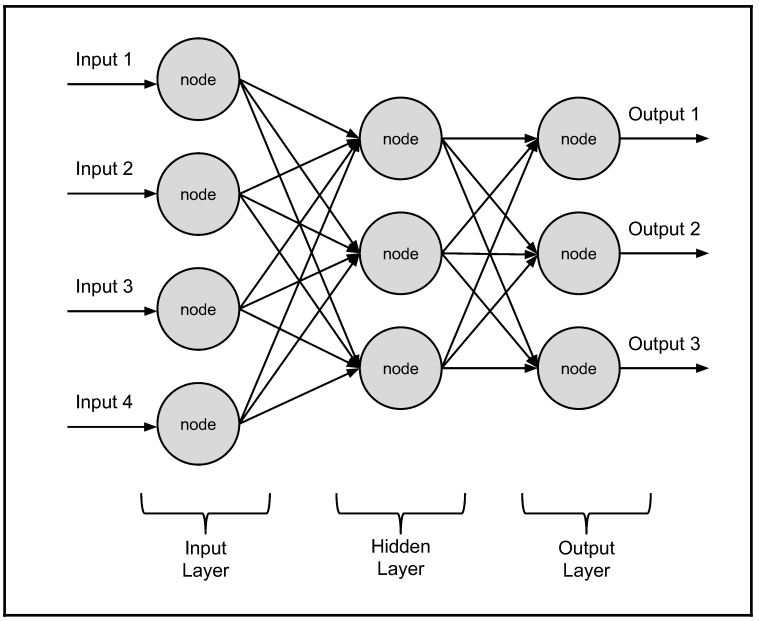
Let's go ahead and define our activation function in Go along with its derivative. These definitions are shown in the following code:

```
// sigmoid implements the sigmoid function
// for use in activation functions.
func sigmoid(x float64) float64 {
    return 1.0 / (1.0 + math.Exp(-x))
}
```

```
// sigmoidPrime implements the derivative
// of the sigmoid function for backpropagation.
func sigmoidPrime(x float64) float64 {
    return x * (1.0 - x)
}
```

Network architecture

The simple neural network that we are going to build will contain an input and output layer (as does any neural network). The network will include a single hidden layer between the input and output layer. This architecture is depicted as follows:



In particular, we are going to include four nodes in the input layer, three nodes in the hidden layer, and three nodes in the output layer. The four nodes in the input layer correspond to the number of attributes that we are going to feed into the network. Think about these four inputs as something like the four measurements we used to classify iris flowers in `Chapter 5`, *Classification*. The output layer will have three nodes because we will be setting up our network to make classifications for the iris flowers, which could be in one of three classes.

Now, regarding the hidden layer--why are we using one hidden layer with three nodes? Well, one hidden layer is sufficient for a very large majority of simple tasks. If you have a large amount of non-linearity in your data, many inputs, and/or large amounts of training data, you may need to introduce more hidden layers (as further discussed later in this chapter in relation to deep learning). The choice of three nodes in the hidden layer can be tuned based on an evaluation metric and a little trial and error. You can also search the numbers of nodes in the hidden layer to automate your choice.



Keep in mind that the more nodes you introduce into the hidden layer, the more perfectly your network can learn your training set. In other words, you are putting yourself in danger of overfitting your model such that it does not generalize. When adding this complication, it is important to validate your model very carefully to ensure that it generalizes.

Why do we expect this architecture to work?

Let's pause for a moment and think about why setting up a series of nodes in the preceding arrangement would allow us to predict something. As you can see, all we are doing is combining inputs over and over to produce some result. How can we expect this result to be useful in making binary classifications?

Well, what exactly do we mean when we define a binary classification problem, or any classification or regression problem, for that matter? We are basically saying that we expect there to be some rules or relationships that exist between a series of inputs (our attributes) and an output (our labels or response). In essence, we are saying that there exists some simple or complicated function that is able to transform our attributes into the response or labels that we are after.

We could make a choice for a type of function that might model this transformation of attributes to output, as we do in linear or logistic regression. However, we could also set up a series of chained and adjustable functions that we could algorithmically train to detect the relationships and rules between our inputs and output. This is what we are doing with the neural network!

The nodes of the neural network are like sub-functions that are adjusted until they mimic the rules and relationships between our supplied inputs and the output, regardless of what those rules and relationships actually are (linear, non-linear, dynamic, and so on). If underlying rules exist between the inputs and outputs, it is likely that there exists some neural network that can mimic the rules.

Training our neural network

Okay, so now we have some good motivation for why this combination of nodes might help us make predictions. How are we actually going to adjust all of the sub-functions of our neural network nodes based on some input data? The answer is called **backpropagation**.

Backpropagation is a method for training our neural network that involves doing the following iteratively over a series of **epochs** (or exposure to our training dataset):

- Feeding our training data forward through the neural network to calculate an output
- Calculating errors in the output
- Using gradient descent (or other relevant method) to determine how we should change our weights and biases based on the errors
- Backpropagating these weight/bias changes into the network

We will implement this process later and go through some of the details. However, certain things like gradient descent are covered in more detail in the `Appendix`, *Algorithms/Techniques Related to Machine Learning*.



This backpropagation method of training a neural network is ubiquitous in the modeling world, but there are tons of unique network architectures and methods that we do not have space to cover here. Neural nets are an active area of research in academia and industry.

First, let's define a struct `neuralNetConfig` that will contain all of the parameters that define our network architecture and how we will go about our backpropagation iterations. Let's also define a struct `neuralNet` that will contain all of the information that defines a trained neural network. Later on, we will utilize trained `neuralNet` values to make predictions. These definitions are shown in the following code:

```
// neuralNet contains all of the information
// that defines a trained neural network.
type neuralNet struct {
    config neuralNetConfig
    wHidden *mat.Dense
    bHidden *mat.Dense
    wOut *mat.Dense
    bOut *mat.Dense
}

// neuralNetConfig defines our neural network
// architecture and learning parameters.
type neuralNetConfig struct {
```

```
    inputNeurons int
    outputNeurons int
    hiddenNeurons int
    numEpochs int
    learningRate float64
}
```

Here, `wHidden` and `bHidden` are the weights and biases for the hidden layer of the network and `wOut` and `bOut` are the weights and biases for the output layer of the network, respectively. Note that we are using `gonum.org/v1/gonum/mat` matrices for all the weights and biases, and we will use similar matrices to represent our inputs and outputs. This will let us easily perform the operations related to the backpropagation and generalize our training to any number of nodes in the input, hidden, and output layers.

Next, let's define a function that initializes a new neural network based on a `neuralNetConfig` value and a method that trains a `neuralNet` value based on a matrix of inputs (x) and a matrix of labels (y):

```
// NewNetwork initializes a new neural network.
func newNetwork(config neuralNetConfig) *neuralNet {
        return &neuralNet{config: config}
}

// Train trains a neural network using backpropagation.
func (nn *neuralNet) train(x, y *mat.Dense) error {

    // To be filled in...
}
```

In the `train()` method, we need to complete our backpropagation method and place the resulting trained weights and biases into the receiver. First, let's initialize the weights and biases randomly in the `train()` method, as shown in the following code:

```
// Initialize biases/weights.
randSource := rand.NewSource(time.Now().UnixNano())
randGen := rand.New(randSource)

wHiddenRaw := make([]float64,
nn.config.hiddenNeurons*nn.config.inputNeurons)
bHiddenRaw := make([]float64, nn.config.hiddenNeurons)
wOutRaw := make([]float64, nn.config.outputNeurons*nn.config.hiddenNeurons)
bOutRaw := make([]float64, nn.config.outputNeurons)

for _, param := range [][]float64{wHiddenRaw, bHiddenRaw, wOutRaw, bOutRaw}
{
    for i := range param {
        param[i] = randGen.Float64()
```

```
        }
    }

    wHidden := mat.NewDense(nn.config.inputNeurons, nn.config.hiddenNeurons,
    wHiddenRaw)
    bHidden := mat.NewDense(1, nn.config.hiddenNeurons, bHiddenRaw)
    wOut := mat.NewDense(nn.config.hiddenNeurons, nn.config.outputNeurons,
    wOutRaw)
    bOut := mat.NewDense(1, nn.config.outputNeurons, bOutRaw)
```

Then, we need to loop over each of our epochs completing the backpropagation of the network. This again involves a feed forward stage in which outputs are calculated and a backpropagating stage in which changes to the weights and biases are applied. The backpropagation process will be discussed in more detail in the `Appendix`, *Algorithms/Techniques Related to Machine Learning*, if you are interested in diving a little deeper. For now, let's focus on the implementation. The loop over epochs look like the following, with the various pieces of the backpropagation process indicated by comments:

```
    // Define the output of the neural network.
    output := mat.NewDense(0, 0, nil)

    // Loop over the number of epochs utilizing
    // backpropagation to train our model.
    for i := 0; i < nn.config.numEpochs; i++ {

        // Complete the feed forward process.

        ...

        // Complete the backpropagation.

        ...

        // Adjust the parameters.

        ...
    }
```

In the feed forward section of this loop, the inputs are propagated forward through our network of nodes:

```
    // Complete the feed forward process.
    hiddenLayerInput := mat.NewDense(0, 0, nil)
    hiddenLayerInput.Mul(x, wHidden)
    addBHidden := func(_, col int, v float64) float64 { return v +
    bHidden.At(0, col) }
    hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)
```

```
hiddenLayerActivations := mat.NewDense(0, 0, nil)
applySigmoid := func(_, _ int, v float64) float64 { return sigmoid(v) }
hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)

outputLayerInput := mat.NewDense(0, 0, nil)
outputLayerInput.Mul(hiddenLayerActivations, wOut)
addBOut := func(_, col int, v float64) float64 { return v + bOut.At(0, col)
}
outputLayerInput.Apply(addBOut, outputLayerInput)
output.Apply(applySigmoid, outputLayerInput)
```

Then, once we have an output from the feed forward process, we go backward through the network calculating deltas (or changes) for the output and hidden layers, as can be seen in the following code:

```
// Complete the backpropagation.
networkError := mat.NewDense(0, 0, nil)
networkError.Sub(y, output)

slopeOutputLayer := mat.NewDense(0, 0, nil)
applySigmoidPrime := func(_, _ int, v float64) float64 { return
sigmoidPrime(v) }
slopeOutputLayer.Apply(applySigmoidPrime, output)
slopeHiddenLayer := mat.NewDense(0, 0, nil)
slopeHiddenLayer.Apply(applySigmoidPrime, hiddenLayerActivations)

dOutput := mat.NewDense(0, 0, nil)
dOutput.MulElem(networkError, slopeOutputLayer)
errorAtHiddenLayer := mat.NewDense(0, 0, nil)
errorAtHiddenLayer.Mul(dOutput, wOut.T())

dHiddenLayer := mat.NewDense(0, 0, nil)
dHiddenLayer.MulElem(errorAtHiddenLayer, slopeHiddenLayer)
```

The deltas are then utilized to update the weights and biases of our network. A **learning rate** is also used to scale these changes, which can help the algorithm converge. This final piece of the backprogation loop is implemented here:

```
// Adjust the parameters.
wOutAdj := mat.NewDense(0, 0, nil)
wOutAdj.Mul(hiddenLayerActivations.T(), dOutput)
wOutAdj.Scale(nn.config.learningRate, wOutAdj)
wOut.Add(wOut, wOutAdj)

bOutAdj, err := sumAlongAxis(0, dOutput)
if err != nil {
    return err
}
```

```
bOutAdj.Scale(nn.config.learningRate, bOutAdj)
bOut.Add(bOut, bOutAdj)

wHiddenAdj := mat.NewDense(0, 0, nil)
wHiddenAdj.Mul(x.T(), dHiddenLayer)
wHiddenAdj.Scale(nn.config.learningRate, wHiddenAdj)
wHidden.Add(wHidden, wHiddenAdj)

bHiddenAdj, err := sumAlongAxis(0, dHiddenLayer)
if err != nil {
    return err
}
bHiddenAdj.Scale(nn.config.learningRate, bHiddenAdj)
bHidden.Add(bHidden, bHiddenAdj)
```

Note, here we have utilized another helper function that allows us to sum matrices along one dimension while keeping the other dimension intact. This `sumAlongAxis()` function is shown in the following code for completeness:

```
// sumAlongAxis sums a matrix along a
// particular dimension, preserving the
// other dimension.
func sumAlongAxis(axis int, m *mat.Dense) (*mat.Dense, error) {

    numRows, numCols := m.Dims()

    var output *mat.Dense

    switch axis {
    case 0:
        data := make([]float64, numCols)
        for i := 0; i < numCols; i++ {
            col := mat.Col(nil, i, m)
            data[i] = floats.Sum(col)
        }
        output = mat.NewDense(1, numCols, data)
    case 1:
        data := make([]float64, numRows)
        for i := 0; i < numRows; i++ {
            row := mat.Row(nil, i, m)
            data[i] = floats.Sum(row)
        }
        output = mat.NewDense(numRows, 1, data)
    default:
        return nil, errors.New("invalid axis, must be 0 or 1")
    }

    return output, nil
```

```
}
```

The last thing that we will do in the `train()` method is add the trained weights and biases to the receiver value and return:

```
// Define our trained neural network.
nn.wHidden = wHidden
nn.bHidden = bHidden
nn.wOut = wOut
nn.bOut = bOut

return nil
```

Awesome! That wasn't so bad. We have a method to train our neural network in about 100 lines of Go code. To check whether this code runs and to get a sense about what our weights and biases might look like, let's train a neural network on some simple dummy data. We will perform a more realistic example with the network in the next section, but for now, let's just sanity check ourselves with some dummy data, as shown in the following code:

```
// Define our input attributes.
input := mat.NewDense(3, 4, []float64{
    1.0, 0.0, 1.0, 0.0,
    1.0, 0.0, 1.0, 1.0,
    0.0, 1.0, 0.0, 1.0,
})

// Define our labels.
labels := mat.NewDense(3, 1, []float64{1.0, 1.0, 0.0})

// Define our network architecture and
// learning parameters.
config := neuralNetConfig{
    inputNeurons: 4,
    outputNeurons: 1,
    hiddenNeurons: 3,
    numEpochs: 5000,
    learningRate: 0.3,
}

// Train the neural network.
network := newNetwork(config)
if err := network.train(input, labels); err != nil {
    log.Fatal(err)
}

// Output the weights that define our network!
f := mat.Formatted(network.wHidden, mat.Prefix(" "))
```

```
fmt.Printf("\nwHidden = % v\n\n", f)

f = mat.Formatted(network.bHidden, mat.Prefix(" "))
fmt.Printf("\nbHidden = % v\n\n", f)

f = mat.Formatted(network.wOut, mat.Prefix(" "))
fmt.Printf("\nwOut = % v\n\n", f)

f = mat.Formatted(network.bOut, mat.Prefix(" "))
fmt.Printf("\nbOut = % v\n\n", f)
```

Compiling and running this neural network training code results in the following output weights and biases:

```
bOut$ go build
$ ./myprogram

wHidden = [ 2.105009524680073 2.022752980359344 2.5200192466310547]
          [ -2.033896626042324 -1.8652059633980662 -1.5861504822640748]
          [1.821046795894909 2.436963623058306 1.7175100168873831]
          [-0.7400175664445425 -0.5800261988090052 -0.9277709997772002]


bHidden = [ 0.06850421921273529 -0.40252869229501825 -0.03392255287230178]


wOut = [ 2.321901257946553]
       [ 3.343178681830189]
       [1.7581701319375231]


bOut = [-3.4716133339240471]
```

These weights and biases fully define our trained neural network. That is, `wHidden` and `bHidden` allow us to translate our inputs into values that are input to the output layer of our network, and `wOut` and `bOut` allow us to translate those values into the final output of our neural net.



Because of our use of random numbers, your program may return slightly different results from those which are described earlier. However, you should see a similar range of values.

Utilizing the simple neural network

Now that we have some neural network training functionality that appears to be working, let's try to utilize this functionality in a more realistic modeling scenario. In particular, let's bring back our favorite classification dataset, the iris flower dataset (utilized in `Chapter 5`, *Classification*).

If you remember, when trying to classify iris flowers using this dataset, we are trying to classify them into one of three species (setosa, virginica, or versicolor). As our neural net is expecting matrices of float values, we need to encode the three species into numerical columns. One way to do this is to create a column in our dataset for each species. We will then set that column's values to either *1.0* or *0.0* depending on whether the corresponding row's measurements correspond to that species (*1.0*) or to another species (*0.0*).

We are also going to standardize our data for reasons similar to those that were discussed in reference to logistic regression in `Chapter 5`, *Classification*. This makes our dataset look slightly different than it did for previous examples, as you can see here:

```
$ head train.csv
sepal_length,sepal_width,petal_length,petal_width,setosa,virginica,versicol
or
0.305555555556,0.583333333333,0.118644067797,0.0416666666667,1.0,0.0,0.0
0.25,0.291666666667,0.491525423729,0.541666666667,0.0,0.0,1.0
0.194444444444,0.5,0.0338983050847,0.0416666666667,1.0,0.0,0.0
0.833333333333,0.375,0.898305084746,0.708333333333,0.0,1.0,0.0
0.555555555556,0.208333333333,0.661016949153,0.583333333333,0.0,0.0,1.0
0.416666666667,0.25,0.508474576271,0.458333333333,0.0,0.0,1.0
0.666666666667,0.416666666667,0.677966101695,0.666666666667,0.0,0.0,1.0
0.416666666667,0.291666666667,0.491525423729,0.458333333333,0.0,0.0,1.0
0.5,0.416666666667,0.661016949153,0.708333333333,0.0,1.0,0.0
```

We will use an 80/20 training and test split of our data to evaluate the model. The training data will be stored in `train.csv` and the testing data will be stored in `test.csv`, as shown here:

```
$ wc -l *.csv
  31 test.csv
 121 train.csv
 152 total
```

Training the neural network on real data

To train our neural network on this iris data, we need to read in the training data and create two matrices. The first matrix will hold all of the attributes (matrix `inputs`), and the second matrix will hold all the labels (matrix `labels`). We are going to construct these matrices as follows:

```
// Open the training dataset file.
f, err := os.Open("train.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 7

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// inputsData and labelsData will hold all the
// float values that will eventually be
// used to form our matrices.
inputsData := make([]float64, 4*len(rawCSVData))
labelsData := make([]float64, 3*len(rawCSVData))

// inputsIndex will track the current index of
// inputs matrix values.
var inputsIndex int
var labelsIndex int

// Sequentially move the rows into a slice of floats.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {
        continue
    }

    // Loop over the float columns.
    for i, val := range record {

        // Convert the value to a float.
```

```
            parsedVal, err := strconv.ParseFloat(val, 64)
            if err != nil {
                log.Fatal(err)
            }

            // Add to the labelsData if relevant.
            if i == 4 || i == 5 || i == 6 {
                labelsData[labelsIndex] = parsedVal
                labelsIndex++
                continue
            }

            // Add the float value to the slice of floats.
            inputsData[inputsIndex] = parsedVal
            inputsIndex++
        }
    }

    // Form the matrices.
    inputs := mat.NewDense(len(rawCSVData), 4, inputsData)
    labels := mat.NewDense(len(rawCSVData), 3, labelsData)
```

We can then initialize and train our neural network, similar to how we did with the dummy data. The only difference is that we are going to use three output neurons corresponding to our three output classes. The training process is as follows:

```
    // Define our network architecture and
    // learning parameters.
    config := neuralNetConfig{
        inputNeurons:  4,
        outputNeurons: 3,
        hiddenNeurons: 3,
        numEpochs:     5000,
        learningRate:  0.3,
    }

    // Train the neural network.
    network := newNetwork(config)
    if err := network.train(inputs, labels); err != nil {
        log.Fatal(err)
    }
```

Evaluating the neural network

To make predictions with the trained neural network, we will create a `predict()` method for the `neuralNet` type. This `predict` method will take in new inputs and complete and feed the new inputs forward through the network to produce predicted outputs, as shown here:

```
// predict makes a prediction based on a trained
// neural network.
func (nn *neuralNet) predict(x *mat.Dense) (*mat.Dense, error) {

    // Check to make sure that our neuralNet value
    // represents a trained model.
    if nn.wHidden == nil || nn.wOut == nil || nn.bHidden == nil || nn.bOut
== nil {
        return nil, errors.New("the supplied neural net weights and biases
are empty")
    }

    // Define the output of the neural network.
    output := mat.NewDense(0, 0, nil)
github.com/tensorflow/tensorflow/tensorflow/go
    // Complete the feed forward process.
    hiddenLayerInput := mat.NewDense(0, 0, nil)
    hiddenLayerInput.Mul(x, nn.wHidden)
    addBHidden := func(_, col int, v float64) float64 { return v +
nn.bHidden.At(0, col) }
    hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)

    hiddenLayerActivations := mat.NewDense(0, 0, nil)
    applySigmoid := func(_, _ int, v float64) float64 { return sigmoid(v) }
    hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)

    outputLayerInput := mat.NewDense(0, 0, nil)
    outputLayerInput.Mul(hiddenLayerActivations, nn.wOut)
    addBOut := func(_, col int, v float64) float64 { return v +
nn.bOut.At(0, col) }
    outputLayerInput.Apply(addBOut, outputLayerInput)
    output.Apply(applySigmoid, outputLayerInput)

    return output, nil
}
```

We can then use this `predict()` method on the testing data to evaluate our trained model. Specifically, we will read in the data in `test.csv` into two new matrices `testInputs` and `testLabels`, similar to how we read in `train.csv` (as such, we will not include the details here). `testInputs` can be supplied to the `predict()` method to get our predictions and then we can compare our predictions and `testLabels` to calculate an evaluation metric. In this case, we will calculate the accuracy of our model.

One detail that we will take care of as we calculate the accuracy is determining a single output prediction from our model. The network actually produces one output between 0.0 and 1.0 for each of the iris species. We will take the highest of these as the predicted species.

The calculation of accuracy for our model is shown in the following code:

```
// Make the predictions using the trained model.
predictions, err := network.predict(testInputs)
if err != nil {
    log.Fatal(err)
}

// Calculate the accuracy of our model.
var truePosNeg int
numPreds, _ := predictions.Dims()
for i := 0; i < numPreds; i++ {

    // Get the label.
    labelRow := mat.Row(nil, i, testLabels)
    var species int
    for idx, label := range labelRow {
        if label == 1.0 {
            species = idx
            break
        }
    }

    // Accumulate the true positive/negative count.
    if predictions.At(i, species) == floats.Max(mat.Row(nil, i,
predictions)) {
        truePosNeg++
    }
}

// Calculate the accuracy (subset accuracy).
accuracy := float64(truePosNeg) / float64(numPreds)

// Output the Accuracy value to standard out.
fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

Compiling and running the evaluation yields something similar to the following:

```
$ go build
$ ./myprogram

Accuracy = 0.97
```

Yeah! That's pretty good. 97% accuracy in our predictions for the iris flower species with our from-scratch neural network. As mentioned earlier, these single hidden layer neural networks are pretty powerful when it comes to a majority of classification tasks, and you can see that the underlying principles are not really that complicated.

Introducing deep learning

Although simple neural networks, like the one utilized in the preceding section, are extremely powerful for many scenarios, deep neural network architectures have been applied across industries in recent years on various types of data. These more complicated architectures have been used to beat champions at board/video games, drive cars, generate art, transform images, and much more. It almost seems like you can throw anything at these models and they will do something interesting, but they seem to be particularly well-suited for computer vision, speech recognition, textual inference, and other very complicated and hard-to-define tasks.

We are going to introduce deep learning here and run a deep learning model in Go. However, the application and diversity of deep learning models is huge and growing every day.

There are many books and tutorials on the subject, so if this brief introduction sparks your interest, we recommend that you look into one of the following resources:

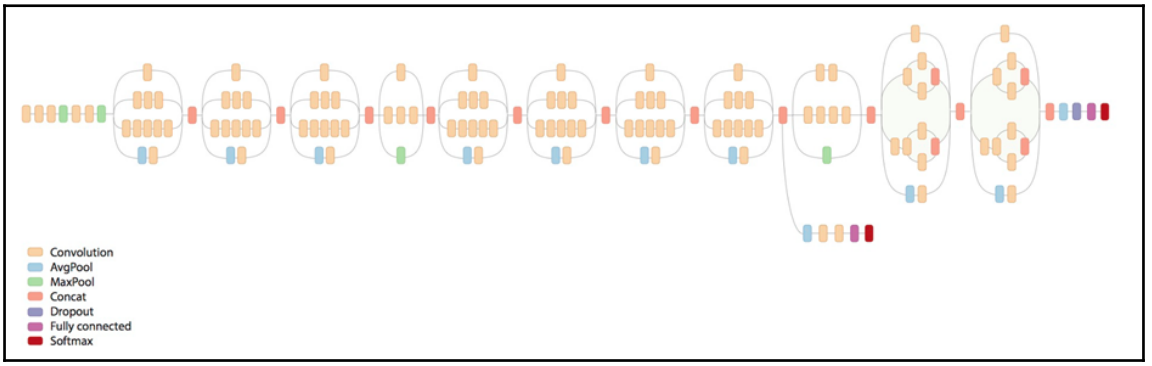
- Hands-On Deep Learning with TensorFlow: `https://www.packtpub.com/big-data-and-business-intelligence/hands-deep-learning-tensorflow`
- Deep Learning by Example: `https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-example`
- Deep Learning for Computer Vision: `https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-computer-vision`

These resources might not all mention or acknowledge Go but, as you will see in the following example, Go is perfectly capable of applying these techniques and interfacing with things like TensorFlow or H2O.

What is a deep learning model?

The deep in deep learning refers to the successive combination and utilization of various neural network structures to form an architecture that is large and complicated. These large and complicated architectures generally require large amounts of data to train, and the resulting structure is very hard to interpret.

To give an example of the scale and complexity of modern deep learning models, take Google's LeNet (`http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf`) as an example. This model, which can be utilized for object recognition, is depicted here (image courtesy of `https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html`):



This structure is hard to digest in and of itself. However, it is made even more impressive when you realize that each of the blocks in this diagram can be a complicated, neural network primitive in and of itself (as shown in the legend).

Generally, deep learning models are constructed by chaining together and, in some cases, rechaining together, one or more of the following structures:

- A fully connected neural network architecture: This building block might be something similar to what we built previously in this chapter for iris flower detection. It includes a number of neural network nodes arranged in fully connected layers. That is, the layers flow from one to the other and all the nodes are connected from one layer to the next.

- A **convolutional neural network** (**ConvNet** or **CNN**): This is a neural network that implements at least one **convolutional layer**. A convolutional layer contains neurons parameterized by a set of weights (also called a convolutional kernel or filter) that is only partially connected to the input data. Most ConvNets include a hierarchical combination of these convolutional layers allows a ConvNet to respond to low level features in the input data.
- A **recurrent neural network** and/or **Long Short-Term Memory** (**LSTM**) **cells**: These components attempt to factor in something like memory into the model. Recurrent neural nets make their output dependent on a sequence of inputs rather than assuming that every input is independent. LSTM cells are related to recurrent neural networks but also try to factor in a longer term sort of memory.



Although we will utilize image data to illustrate deep learning in this chapter, deep learning models using recurrent neural nets and LSTMs are particularly well-adapted at solving text-based and other NLP problems. The memory that is built into these networks allows them to understand what words might come next in a sentence, for example. If you are interested in running a deep learning model that is based on text data, we recommend looking at this example that uses the `github.com/chewxy/gorgonia` package: `https://github.com/chewxy/gorgonia/tree/master/examples/charRNN`.



Deep learning models are powerful! Especially for tasks like computer vision. However, you should also keep in mind that complicated combinations of these neural net components are also extremely hard to interpret. That is, determining why the model made a certain prediction can be near impossible. This can be a problem when you need to maintain compliance in certain industries and jurisdictions, and it also might inhibit debugging or maintenance of your applications. That being said, there are some major efforts to improve the interpretability of deep learning models. Notable among these efforts is the LIME project: `https://github.com/marcotcr/lime`.

Deep learning with Go

There are a variety of options when you are looking to build or utilize deep learning models from Go. This, as with deep learning itself, is an ever changing landscape. However, the options for building, training and utilizing deep learning models in Go are generally as follows:

- **Use a Go package**: There are Go packages that allow you to use Go as your main interface to build and train deep learning models. The most features and developed of these packages is `github.com/chewxy/gorgonia`. `github.com/chewxy/gorgonia` treats Go as a first-class citizen and is written in Go, even if it does make significant usage of `cgo` to interface with numerical libraries.
- **Use an API or Go client for a non-Go DL framework**: You can interface with popular deep learning services and frameworks from Go including TensorFlow, MachineBox, H2O, and the various cloud providers or third-party API offerings (such as IBM Watson). TensorFlow and Machine Box actually have Go bindings or SDKs, which are continually improving. For the other services, you may need to interact via REST or even call binaries using `exec`.
- **Use cgo**: Of course, Go can talk to and integrate with C/C++ libraries for deep learning, including the TensorFlow libraries and various libraries from Intel. However, this is a difficult road, and it is only recommended when absolutely necessary.

As TensorFlow is by far the most popular framework for deep learning (at the moment), we will briefly explore the second category listed here. However, the Tensorflow Go bindings are under active development and some functionality is quite crude at the moment. The TensorFlow team recommends that if you are going to use a TensorFlow model in Go, you first train and export this model using Python. That pre-trained model can then be utilized from Go, as we will demonstrate in the next section.

There are a number of members of the community working very hard to make Go more of a first-class citizen for TensorFlow. As such, it is likely that the rough edges of the TensorFlow bindings will be smoothed over the coming year. To keep updated on this front, make sure and join the #data-science channel on Gophers Slack (`https://invite.slack.golangbridge.org/`). This is a frequent topic of conversation there and would be a great place to ask questions and get involved.



Even though we will explore a quick TensorFlow example here, we highly recommend that you should look into `github.com/chewxy/gorgonia`, especially if you are considering doing more custom or extensive modeling in Go. This package is super powerful and is used in production.

Setting up TensorFlow for use with Go

The TensorFlow team has provided some good docs to install TensorFlow and get it ready for usage with Go. These docs can be found at `https://www.tensorflow.org/install/install_go`. There are a couple of preliminary steps, but once you have the TensorFlow C libraries installed, you can get the following Go package:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
```

Everything should be good to go if you were able to get `github.com/tensorflow/tensorflow/tensorflow/go` without error, but you can make sure that you are ready to use TensorFlow by executing the following tests:

```
$ go test github.com/tensorflow/tensorflow/tensorflow/go
ok github.com/tensorflow/tensorflow/tensorflow/go 0.045s
```

Retrieving and calling a pretrained TensorFlow model

The model that we are going to use is a Google model for object recognition in images called Inception. The model can be retrieved as follows:

```
$ mkdir model
$ cd model
$ wget
https://storage.googleapis.com/download.tensorflow.org/models/inception5h.z
ip
--2017-09-09 18:29:03--
https://storage.googleapis.com/download.tensorflow.org/models/inception5h.z
ip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.6.112,
2607:f8b0:4009:812::2010
Connecting to storage.googleapis.com
(storage.googleapis.com)|172.217.6.112|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 49937555 (48M) [application/zip]
Saving to: 'inception5h.zip'

inception5h.zip
100%[=====================================================================
===================================================>] 47.62M 19.0MB/s in
2.5s

2017-09-09 18:29:06 (19.0 MB/s) - 'inception5h.zip' saved [49937555/49937555]

$ unzip inception5h.zip
Archive: inception5h.zip
```

```
inflating: imagenet_comp_graph_label_strings.txt
inflating: tensorflow_inception_graph.pb
inflating: LICENSE
```

After unzipping the compressed model, you should see a `*.pb` file. This is a `protobuf` file that represents a frozen state of the model. Think back to our simple neural network. The network was fully defined by a series of weights and biases. Although more complicated, this model can be defined in a similar way and these definitions are stored in this `protobuf` file.

To call this model, we will use some example code from the TensorFlow Go bindings docs--`https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go`. This code loads the model and uses the model to detect and label the contents of a `*.jpg` image.

As the code is included in the TensorFlow docs, I will spare the details and just highlight a couple of snippets. To load the model, we perform the following:

```
// Load the serialized GraphDef from a file.
modelfile, labelsfile, err := modelFiles(*modeldir)
if err != nil {
    log.Fatal(err)
}
model, err := ioutil.ReadFile(modelfile)
if err != nil {
    log.Fatal(err)
}
```

Then we load the graph definition of the deep learning model and create a new TensorFlow session with the graph, as shown in the following code:

```
// Construct an in-memory graph from the serialized form.
graph := tf.NewGraph()
if err := graph.Import(model, ""); err != nil {
    log.Fatal(err)
}

// Create a session for inference over graph.
session, err := tf.NewSession(graph, nil)
if err != nil {
    log.Fatal(err)
}
defer session.Close()
```

Finally, we can make an inference using the model as follows:

```
// Run inference on *imageFile.
// For multiple images, session.Run() can be called in a loop (and
```

```
// concurrently). Alternatively, images can be batched since the model
// accepts batches of image data as input.
tensor, err := makeTensorFromImage(*imagefile)
if err != nil {
    log.Fatal(err)
}
output, err := session.Run(
    map[tf.Output]*tf.Tensor{
        graph.Operation("input").Output(0): tensor,
    },
    []tf.Output{
        graph.Operation("output").Output(0),
    },
    nil)
if err != nil {
    log.Fatal(err)
}

// output[0].Value() is a vector containing probabilities of
// labels for each image in the "batch". The batch size was 1.
// Find the most probable label index.
probabilities := output[0].Value().([][]float32)[0]
printBestLabel(probabilities, labelsfile)
```

Object detection using TensorFlow from Go

The Go program for object detection, as specified in the TensorFlow GoDocs, can be called as follows:

```
$ ./myprogram -dir=<path/to/the/model/dir> -image=<path/to/a/jpg/image>
```

When the program is called, it will utilize the pretrained and loaded model to infer the contents of the specified image. It will then output the most likely contents of that image along with its calculated probability.

To illustrate this, let's try performing the object detection on the following image of an airplane, saved as `airplane.jpg`:



Running the TensorFlow model from Go gives the following results:

```
$ go build
$ ./myprogram -dir=model -image=airplane.jpg
2017-09-09 20:17:30.655757: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655807: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655814: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655818: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655822: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (86% likely) airliner
```

After some suggestions about speeding up CPU computations, we get a result: `airliner`. Wow! That's pretty cool. We just performed object recognition with TensorFlow right from our Go program!

Let try another one. This time, we will use `pug.jpg`, which looks like the following:

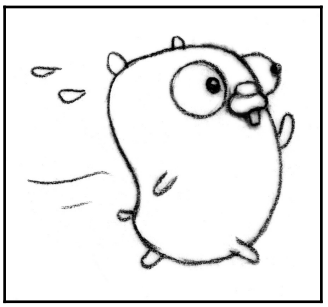


Running our program again with this image gives the following:

```
$ ./myprogram -dir=model -image=pug.jpg
2017-09-09 20:20:32.323855: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323896: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323902: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323906: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323911: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (84% likely) pug
```

Success! Not only did the model detect that there was a dog in the picture, it correctly identified that there was a pug dog in the picture.

Let try just one more. As this is a Go book, we cannot resist trying `gopher.jpg`, which looks like the following (huge thanks to Renee French, the artist behind the Go gopher):



Running the model gives the following result:

```
$ ./myprogram -dir=model -image=gopher.jpg
2017-09-09 20:25:57.967753: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967801: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967808: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967812: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967817: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (12% likely) safety pin
```

Well, I guess we can't win them all. Looks like we need to refactor our model to be able to recognize Go gophers. More specifically, we should probably add a bunch of Go gophers to our training dataset, because a Go gopher is definitely not a safety pin!

References

Basic neural networks:

- A quick introduction to neural networks: `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/`
- A friendly introduction to deep learning and neural networks: `https://youtu.be/BR9h47Jtqyw`
- `github.com/tleyden/neurgo` docs: `https://godoc.org/github.com/tleyden/neurgo`
- `github.com/fxsjy/gonn/gonn` docs: `https://godoc.org/github.com/fxsjy/gonn/gonn`

More detailed deep learning resources:

- Hands-On Deep Learning with TensorFlow - `https://www.packtpub.com/big-data-and-business-intelligence/hands-deep-learning-tensorflow`
- Deep Learning by Example - `https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-example`
- Deep Learning for Computer Vision - `https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-computer-vision`

Deep learning with Go:

- TensorFlow Go bindings docs: `https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go`
- `github.com/chewxy/gorgonia` docs: `https://godoc.org/github.com/chewxy/gorgonia`
- MachineBox Go SDK docs: `https://godoc.org/github.com/machinebox/sdk-go`
- Calling pretrained models using `cgo` example: `https://github.com/gopherdata/dlinfer`

Summary

Congratulations! We have gone from parsing data with Go to calling deep learning models from Go. You now know the basics of neural networks and can implement them and utilize them in your Go programs. In the next chapter, we will discuss how to get these models and applications off of your laptops and run them at production scale in data pipelines.

9
Deploying and Distributing Analyses and Models

We have implemented all sorts of models in Go, including regressions, classifications, clustering, and more. You have also learned about some of the process around developing a machine learning model. Our models have successfully predicted disease progression, flower species, and objects within images. Yet, we are still missing an important piece of the machine learning puzzle: deployment, maintenance, and scaling.

If our models just stay on our laptops, they are not doing any good or creating value within a company. We need to know how to take our machine learning workflows and integrate them into the systems that are already deployed in our organization, and we need to know how to scale, update, and maintain these workflows over time.

The fact that our machine learning workflows are, by their very nature, multi-stage workflows might make this deployment and maintenance a little bit of a challenge. We need to train, test, and utilize our models and, in some cases, we need to preprocess and/or postprocess our data. We may also need to chain certain models together. How can we deploy and connect all of these stages while maintaining the simplicity and integrity of our applications? For example, how can we update a training dataset over time while still knowing which training dataset produced which models, and which of those models produced which results? How can we easily scale our predictions as the demand for those predictions scales up and down? Finally, how can we integrate our machine learning workflows with other applications in our infrastructure or with pieces of the infrastructure itself (databases, queues, and so on)?

We will tackle all of these questions in this chapter. As it turns out, Go and infrastructure tooling written in Go provide an excellent platform to deploy and manage machine learning workflows. We will use a completely Go-based approach from bottom to top and illustrate how each of those pieces helps us do data science at scale!

Running models reliably on remote machines

Whether your company uses on--premise infrastructure or cloud infrastructure, you will need to run your machine learning models somewhere other than your laptop at some point. These models might need to be ready to serve fraud predictions, or they may need to process user-uploaded images in real time. You cannot have the model sitting on your laptop and successfully serve this information.

However, as we get our data processing and machine learning applications off of our laptops, we should ensure the following:

1. We should not complicate our applications just for the sake of deployment and scaling. We should keep our applications simple, which will help us maintain them and ensure integrity over time.
2. We should ensure that our applications behave like they did on our local machine, where we developed them.

If we deploy our machine learning workflows and they do no perform or behave like they did during development, they will not be able to produce their anticipated value. We should be able to understand how our models will perform locally and assume that they will perform the same way in production. This becomes increasingly hard to accomplish as you add unnecessary complexity to your applications during deployment.

One way to keep your deployments simple, portable, and reproducible is with Docker (`https://www.docker.com/`), and we will utilize it here to deploy our machine learning applications.



Docker is itself written in Go, making it the first Go-based infrastructure tool that we will make use of in our machine learning deployment stack.

A brief introduction to Docker and Docker jargon

Docker and the whole container ecosystem has its own set of jargon, which can be confusing, especially for those with experience in things like virtual machines. Before we continue, let's solidify this jargon as follows:

- A **Docker image** is a collection of data layers that together define a filesystem, libraries, environmental variables, and so on that will be seen by your application running inside a software container. Think of the image as a package that includes your application, other related libraries or packages, and other portions of an environment that your applications, need to run. A Docker image does not include a full operating system.
- A **Dockerfile** is a file in which you define the various layers of your Docker image.
- The **Docker engine** helps you build, manage, and run Docker images.
- The process of building a Docker image for your application is commonly referred to as **Docker-izing** your application.
- A **container** or **software container** is a running instance of a Docker image. Essentially, this running container includes all of the layers of your Docker image plus a read/write layer allowing your application to run, input/output data, and so on.
- A **Docker registry** is a place where you keep Docker images. This registry could be local or it could be running on a remote machine. It also could be a hosted registry service such as **Docker Hub** or Quay, **Amazon Web Service** (**AWS**), **Amazon EC2 Container Registry** (**ECR**), and so on.



Note: A Docker image is not the same as a virtual machine. A Docker image includes your application, a filesystem, and various libraries and packages, but it does not actually include a guest operation system. Moreover, it does not take up a set amount of memory, disk, and CPU on the host machine when running. Docker containers share the resources of the underlying kernel on which the Docker engine is running.

We hope that all of this jargon is solidified in the examples that follow this section, but as with other subjects in this book, you can dive into Docker and software containers deeper. We will include some links in this chapter's references to more Docker resources.

In the following examples, we will assume that you are able to build and run Docker images locally. To install Docker, you can follow the detailed instructions at `https://www.docker.com/community-edition#/download`.

Docker-izing a machine learning application

The machine learning workflow that we will be deploying and scaling in this chapter will be the linear regression workflow to predict diabetes disease progression that we developed in `Chapter 4`, *Regression*. In our deployment, we will consider three different pieces of the workflow:

- The training and exporting of a single regression model (modeling disease progression with body mass index)
- The training and exporting of a multiple regression model (modeling disease progression with body mass index and the blood measurement LTG)
- An inference of disease progression based on one of the trained models and input attributes

Later in the chapter, it will become clear why we might want to split the workflow into these pieces. For now, let's focus on getting these pieces of the workflow Docker-ized (building Docker images that can run these portions of our workflow, that is).

Docker-izing the model training and export

We are going to utilize essentially the same code for model training as that from `Chapter 4`, *Regression*. However, we are going to make a few tweaks to the code to make it more user-friendly and able to interface with other portions of our workflow. We would not consider these complications to the actual modeling code. Rather, these are things that you would probably do to any application that you are getting ready to utilize more generally.

First, we are going to add some command line flags to our application that will allow us to specify the input directory where our training dataset will be located, and an output directory where we are going to export a persisted representation of our model. You can implement these command line arguments as follows:

```
// Declare the input and output directory flags.
inDirPtr := flag.String("inDir", "", "The directory containing the training
data")
outDirPtr := flag.String("outDir", "", "The output directory")

// Parse the command line flags.
flag.Parse()
```

Then we will create a couple of struct types that will allow us to export the coefficient and intercept of our model to a JSON file. This exported JSON file is essentially a persisted version of our trained model, because the coefficients and intercept fully parameterize our model. These structs are defined here:

```
// ModelInfo includes the information about the
// model that is output from the training.
type ModelInfo struct {
    Intercept float64 `json:"intercept"`
    Coefficients []CoefficientInfo `json:"coefficients"`
}

// CoefficientInfo include information about a
// particular model coefficient.
type CoefficientInfo struct {
    Name string `json:"name"`
    Coefficient float64 `json:"coefficient"`
}
```

Other than that, our training code is the same as it was from `Chapter 4`, *Regression*. We will still use `github.com/sajari/regression` to train our model. We will just export the model to the JSON file. The training and exporting of the single regression model is included in the following snippet:

```
// Train/fit the single regression model
// with github.com/sajari/regression
// like in Chapter 5, Regression.

// Fill in the model information.
modelInfo := ModelInfo{
    Intercept: r.Coeff(0),
    Coefficients: []CoefficientInfo{
        CoefficientInfo{
            Name: "bmi",
            Coefficient: r.Coeff(1),
        },
    },
}

// Marshal the model information.
outputData, err := json.MarshalIndent(modelInfo, "", " ")
if err != nil {
    log.Fatal(err)
}

// Save the marshalled output to a file, with
// certain permissions (http://permissions-calculator.org/decode/0644/).
```

```
if err := ioutil.WriteFile(filepath.Join(*outDirPtr, "model.json"),
outputData, 0644); err != nil {e model to the JSON fil
    log.Fatal(err)
}
```

Then, for the multiple regression model, the process looks like the following:

```
// Train/fit the multiple regression model
// with github.com/sajari/regression
// like in Chapter 5, Regression.

// Fill in the model information.
modelInfo := ModelInfo{
    Intercept: r.Coeff(0),
    Coefficients: []CoefficientInfo{
        CoefficientInfo{
            Name:        "bmi",
            Coefficient: r.Coeff(1),
        },
        CoefficientInfo{
            Name:        "ltg",
            Coefficient: r.Coeff(2),
        },
    },
}

// Marshal the model information.
outputData, err := json.MarshalIndent(modelInfo, "", "    ")
if err != nil {
    log.Fatal(err)
}

// Save the marshalled output to a file.
if err := ioutil.WriteFile(filepath.Join(*outDirPtr, "model.json"),
outputData, 0644); err != nil {
    log.Fatal(err)
}
```

To Docker-ize these training processes, we need to create a Dockerfile for each of the single regression training program and the multiple regression training program. It turns out, however, that we can use essentially the same Dockerfile for each of these. This Dockerfile, which should be placed in the same directory as our program, looks like this:

```
FROM alpine
ADD goregtrain /
```

Pretty simple, right? Let's explore the purpose of these two lines. Remember how a Docker image is a series of layers that specify the environment in which your application will run? Well, this Dockerfile builds up two of those layers with two Dockerfile commands, `FROM` and `ADD`.

`FROM alpine` specifies that we want our Docker image filesystem, applications, and libraries to be based on the official Alpine Linux Docker image available on Docker Hub. The reason that we are using `alpine` as a base image is that it is a very small Docker image (making it very portable) and it includes a few Linux shell niceties.

`ADD goregtrain /` specifies that we want to add a `goregtrain` file to the `/` directory in the Docker image. This `goregtrain` file is actually the Go binary that we are going to build from our Go code. Thus, all the Dockerfile is saying is that we want to run our Go binary in Alpine Linux.



Unless you are using `cgo` and depend on a bunch of external C libraries, always build your Go binary before building your Docker image and copy this statically linked Go binary into the Docker image. This will speed up your Docker builds and make your Docker images extremely small, which means that you will be able to port them easily to any system and start them super quick.

Now, we need to build our Go binary before we build our Docker image, because we are copying that Go binary into the image. To do this, we will use a `Makefile` that looks like the following:

```
all: compile docker push clean

compile:
        GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregtrain

docker:
        sudo docker build --force-rm=true -t dwhitena/goregtrain:single .

push:
        sudo docker push dwhitena/goregtrain:single

clean:
        rm goregtrain
```

As you can see:

- `make compile` will compile our Go binary for the target architecture and name it `goregtrain`.

- `make docker` will use the Docker engine (via the `docker` CLI) to build an image based on our Dockerfile, and it will **tag** our image, `dwhitena/goregtrain:single`.
- The `dwhitena` portion of the tag specifies the Docker Hub username under which we will store our image (in this case, `dwhitena`), `goregtrain` specifies the name of the image, and `:single` specifies a tagged version of this image.
- Once the image is built, `make push` will push the newly built Docker image to a registry. In this case, it will push it to Docker Hub under the username of `dwhitena` (of course, you could push to any other private or public registry).
- Finally, `make clean` will clean up our binary.

As mentioned, this `Dockerfile` and `Makefile` are the same for both the single and multiple regression models. However, we will utilize different Docker image tags to differentiate the two models. We will utilize `dwhitena/goregtrain:single` for the single regression model and `dwhitena/goregtrain:multi` for the multiple regression model.



In these examples and in the rest of the chapter, you can follow along locally using either the public Docker images under `dwhitena` on Docker Hub, which would not require you to modify the examples printed here. Just note that you will not be able to build and push your own image to `dwhitena` on Docker Hub, as you are not `dwhitena`. Alternatively, you could replace `dwhitena` everywhere in the examples with your own Docker Hub username. This would allow you to build, push, and utilize your own images.

To build, push, and clean up after building the Docker image for either the single or multiple regression models, we can just run `make`, as shown in the following code, which is itself written in Go:

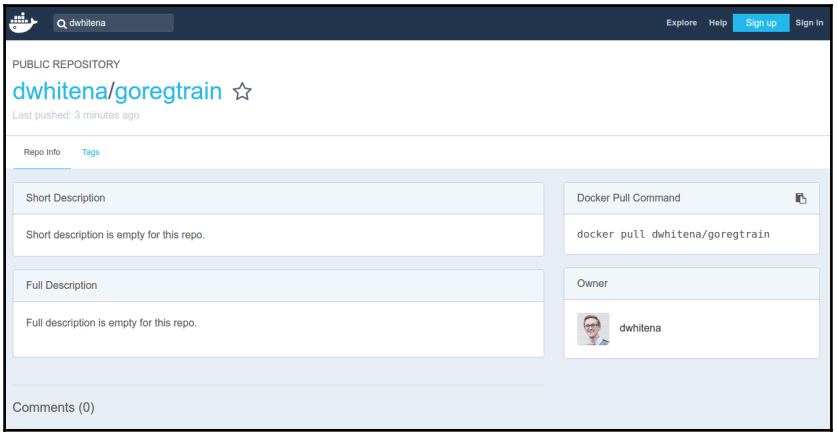
```
$ make
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregtrain
sudo docker build --force-rm=true -t dwhitena/goregtrain:multi .
[sudo] password for dwhitena:
Sending build context to Docker daemon 2.449MB
Step 1/2 : FROM alpine
 ---> 7328f6f8b418
Step 2/2 : ADD goregtrain /
 ---> 9c13594ad7de
Removing intermediate container 6e717232c6d1
Successfully built 9c13594ad7de
Successfully tagged dwhitena/goregtrain:multi
sudo docker push dwhitena/goregtrain:multi
The push refers to a repository [docker.io/dwhitena/goregtrain]
```

```
375642156c06: Pushed
5bef08742407: Layer already exists
multi: digest:
sha256:f35277a46ed840922a0d7e94c3c57632fc947f6ab004deed66d3eb80a331e40f
size: 738
rm goregtrain
```

You can see in the preceding output that the Docker engine built the two layers of our image, tagged the image, and pushed the image up to Docker Hub. We can now see the Docker image in our local registry via the following:

```
$ sudo docker images | grep "goregtrain"
dwhitena/goregtrain multi 9c13594ad7de About a minute ago 6.41MB
```

We can also see it on Docker Hub (`https://hub.docker.com/r/dwhitena/goregtrain/`) as illustrated here:



We will see how to run and utilize this Docker image shortly, but let's first build another Docker image that will generate predictions based on our JSON-persisted models.

Docker-izing model predictions

As with the training of our model, we are going to utilize command line arguments to specify the input directories and output directories utilized by our prediction program. This time we will have two input directories; one for the persisted model and one for a directory that will contain attributes from which we are to make predictions. Our program will, thus, do the following:

1. Read in the model from the model input directory.

2. Walk over files in the attributes input directory.
3. For each file in the attributes input directory (containing attributes with no corresponding prediction of disease progression), utilize our loaded model to make a prediction of disease progression.
4. Output the disease progression to an output file in the directory specified as a command line argument.

Think of this process as follows. We have trained our model on historical data to predict disease progression and we want doctors or clinics to utilize this prediction in some way for new patients. They send us those patients attributes (**body mass index** (**BMI**) or body mass index and **long term growth** (**LTG**)) and we make predictions based on these input attributes.

We will assume that the input attributes come to our program in the form of JSON files (which could also be thought of as a JSON message off a queue or a JSON API response) that look like the following:

```
{
  "independent_variables": [
    {
      "name": "bmi",
      "value": 0.0616962065187
    },
    {
      "name": "ltg",
      "value": 0.0199084208763
    }
  ]
}
```

As such, let's create the following struct types in our prediction program to decode the input attributes and marshal the output prediction:

```
// PredictionData includes the data necessary to make
// a prediction and encodes the output prediction.
type PredictionData struct {
    Prediction float64 `json:"predicted_diabetes_progression"`
    IndependentVars []IndependentVar `json:"independent_variables"`
}

// IndependentVar include information about and a
// value for an independent variable.
type IndependentVar struct {
    Name string `json:"name"`
    Value float64 `json:"value"`
}
```

Let's also create a function that will allow us to make a prediction based on a `ModelInfo` value that we read in from the model input directory (our persisted model, that is). This `prediction` function is shown in the following code:

```
// Predict makes a prediction based on input JSON.
func Predict(modelInfo *ModelInfo, predictionData *PredictionData) error {

    // Initialize the prediction value
    // to the intercept.
    prediction := modelInfo.Intercept

    // Create a map of independent variable coefficients.
    coeffs := make(map[string]float64)
    varNames := make([]string, len(modelInfo.Coefficients))
    for idx, coeff := range modelInfo.Coefficients {
        coeffs[coeff.Name] = coeff.Coefficient
        varNames[idx] = coeff.Name
    }

    // Create a map of the independent variable values.
    varVals := make(map[string]float64)
    for _, indVar := range predictionData.IndependentVars {
        varVals[indVar.Name] = indVar.Value
    }

    // Loop over the independent variables.
    for _, varName := range varNames {

        // Get the coefficient.
        coeff, ok := coeffs[varName]
        if !ok {
            return fmt.Errorf("Could not find model coefficient %s",
varName)
        }

        // Get the variable value.
        val, ok := varVals[varName]
        if !ok {
            return fmt.Errorf("Expected a value for variable %s", varName)
        }

        // Add to the prediction.
        prediction = prediction + coeff*val
    }

    // Add the prediction to the prediction data.
    predictionData.Prediction = prediction
```

```
    return nil
}
```

This prediction function, along with the types, will then allow us to walk over any attribute JSON files in a specified input directory and output disease predictions for each of those files. This process is implemented in the following code:

```
// Declare the input and output directory flags.
inModelDirPtr := flag.String("inModelDir", "", "The directory containing
the model.")
inVarDirPtr := flag.String("inVarDir", "", "The directory containing the
input attributes.")
outDirPtr := flag.String("outDir", "", "The output directory")

// Parse the command line flags.
flag.Parse()

// Load the model file.
f, err := ioutil.ReadFile(filepath.Join(*inModelDirPtr, "model.json"))
if err != nil {
    log.Fatal(err)
}

// Unmarshal the model information.
var modelInfo ModelInfo
if err := json.Unmarshal(f, &modelInfo); err != nil {
    log.Fatal(err)
}

// Walk over files in the input.
if err := filepath.Walk(*inVarDirPtr, func(path string, info os.FileInfo,
err error) error {

    // Skip any directories.
    if info.IsDir() {
        return nil
    }

    // Open any files.
    f, err := ioutil.ReadFile(filepath.Join(*inVarDirPtr, info.Name()))
    if err != nil {
        return err
    }

    // Unmarshal the independent variables.
    var predictionData PredictionData
    if err := json.Unmarshal(f, &predictionData); err != nil {
        return err
```

```
    }

    // Make the prediction.
    if err := Predict(&modelInfo, &predictionData); err != nil {
        return err
    }

    // Marshal the prediction data.
    outputData, err := json.MarshalIndent(predictionData, "", "    ")
    if err != nil {
        log.Fatal(err)
    }

    // Save the marshalled output to a file.
    if err := ioutil.WriteFile(filepath.Join(*outDirPtr, info.Name()),
outputData, 0644); err != nil {
        log.Fatal(err)
    }

    return nil
}); err != nil {
    log.Fatal(err)
}
```

We will use the same `Dockerfile` and `Makefile` that we used in the preceding section to Docker-ize this prediction program. The only difference is that we will name the Go binary `goregpredict` and we will tag the Docker image `dwhitena/goregpredict`. Building the Docker image with `make` should return a similar output to that in the preceding section:

```
$ cat Makefile
all: compile docker push clean

compile:
        GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregpredict

docker:
        sudo docker build --force-rm=true -t dwhitena/goregpredict .

push:
        sudo docker push dwhitena/goregpredict

clean:
        rm goregpredict
$ cat Dockerfile
FROM alpine
ADD goregpredict /
$ make
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregpredict
```

```
sudo docker build --force-rm=true -t dwhitena/goregpredict .
[sudo] password for dwhitena:
Sending build context to Docker daemon 2.38MB
Step 1/2 : FROM alpine
 ---> 7328f6f8b418
Step 2/2 : ADD goregpredict /
 ---> a2d9a63f4926
Removing intermediate container c1610b425835
Successfully built a2d9a63f4926
Successfully tagged dwhitena/goregpredict:latest
sudo docker push dwhitena/goregpredict
The push refers to a repository [docker.io/dwhitena/goregpredict]
77f12cb6c6d4: Pushed
5bef08742407: Layer already exists
latest: digest:
sha256:9a8a754c434bf2250b2f6102bb72d56fdf723f305aebcbf5bff7e5de707dd384
size: 738
3a05f65b1d1d: Layer already exists
5bef08742407: Layer already exists
ult: digest:
sha256:153adaa9b4b9a1f2bf02466201163c60230ae85164d9d22261f455979a94aed4
size: 738
rm goregpredict
```

Testing the Docker images locally

Before pushing our Docker-ized modeling processes to any servers, it's wise to test them locally to ensure that we are seeing the behavior that we expect. Then, once we are satisfied with that behavior, we can rest assured that these Docker images will run exactly the same on any other host that is running Docker. This assurance makes the use of Docker images a significant contributor to maintaining reproducibility with our deploys.

Let's suppose that we have our training data and some input attribute files in the following directories:

```
$ ls
attributes model training
$ ls model
$ ls attributes
1.json 2.json 3.json
$ ls training
diabetes.csv
```

We can run our Docker images as software containers locally using the `docker run` command. We will also utilize the `-v` flag that will let us mount local directories inside of a running container, allowing us to read and write files to and from our local filesystem.

First, let's run our single regression model training inside of the Docker container as follows:

```
$ sudo docker run -v $PWD/training:/tmp/training -v $PWD/model:/tmp/model
dwhitena/goregtrain:single /goregtrain -inDir=/tmp/training -
outDir=/tmp/model

Regression Formula:
Predicted = 152.13 + bmi*949.44
```

Now, if we look at what is in our `model` directory, we will see the newly trained model coefficients and intercept in `model.json`, which was output from our program running in the Docker image. This is illustrated in the following code:

```
$ ls model
model.json
$ cat model/model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

Excellent! We trained our model using the Docker container. Now let's utilize this model to make predictions. Specifically, let's run our `goregpredict` Docker image using the trained model to make predictions for the three attribute files in `attributes`. In the following code, you will see that the attribute files do not have corresponding predictions before running the Docker image, but they do after running the Docker image:

```
$ cat attributes/1.json
{
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
```

```
        }
    ]
}
$ sudo docker run -v $PWD/attributes:/tmp/attributes -v
$PWD/model:/tmp/model dwhitena/goregpredict /goregpredict -
inModelDir=/tmp/model -inVarDir=/tmp/attributes -outDir=/tmp/attributes
$ cat attributes/1.json
{
    "predicted_diabetes_progression": 210.7100380636843,
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
        }
    ]
}
```

Running the Docker images on remote machines

You might be thinking, *what's the big deal, you get the same functionality with Docker that we could get by just building and running your Go program*. Well, that's true locally, but the magic of Docker happens when we want to run our same functionality in an environment other than our laptop.

We can take those Docker images and run them in the exact same way on any host that is running Docker, and they will produce the same exact behavior. This is true for any Docker image. You don't have to install any of the dependencies that might be layered inside of the Docker image. All you need is Docker.

Guess what? Our Docker images are just about 3 MB in size! This means that they will download to any host super fast and start up and run super fast. You don't have to worry about lugging around multiple gigabyte-sized virtual machines and manually specifying resources.

Building a scalable and reproducible machine learning pipeline

Docker sets up quite a bit of the way towards having our machine learning workflows deployed in our company's infrastructure. However, there are still a few missing pieces, as outlined here:

- How do we string the various stages of our workflow together? In this simple example, we have a training stage and a prediction stage. In other pipelines, you might also have data preprocessing, data splitting, data combining, visualization, evaluation, and so on.
- How to get the right data to the right stages of our workflow, especially as we receive new data and/or our data changes? It's not sustainable to manually copy new attributes over to a folder that is co-located with our prediction image every time we need to make new predictions, and we cannot log in to a server every time we need to update our training set.
- How will we be able to track and reproduce the various runs of our workflow for maintenance, continued development, or debugging? If we are making predictions over time and updating our model and/or training set over time, we need to understand which models produced which results to maintain reproducibility (and to meet compliance, in some cases).
- How can we scale our processing over multiple machines and, in some cases, over multiple shared resources? We likely need to run our processing on some shared set of resources in our company. It would be nice for us to be able to scale our processing over these resources as we need more computation power and/or as other applications are scheduled on these resources.

Thankfully, there are a couple more open source tools (both written in Go) that solve these issues for us. Not only that, they let us use the Docker images that we already built as the main units of data processing. These tools are **Kubernetes** (**k8s**) and **Pachyderm**.

You have already been exposed to Pachyderm in `Chapter 1`, *Gathering and Organizing Data*. In that chapter, we utilized Pachyderm to illustrate the idea of data versioning and, as you might be guessing, Pachyderm will help us solve some of the issues around managing data, tracking, and reproducibility. Pachyderm will also solve all of the remaining issues that we need to address around scalability and pipelining because Pachyderm provides both data management/versioning capabilities and data pipelining capabilities.

Kubernetes is a container orchestration engine, which is amazing at scheduling containerized workloads (like our Docker images) across a cluster of shared resources. It is wildly popular right now, and Pachyderm utilizes Kubernetes under the hood to manage containerized data processing pipelines.

Setting up a Pachyderm and Kubernetes cluster

Pachyderm, which runs on Kubernetes, can be deployed almost anywhere because Kubernetes can be deployed anywhere. You can deploy Pachyderm on any of the popular cloud providers, on premise, or even on your own laptop. All you need to do to deploy a Pachyderm cluster is the following:

1. Have a running Kubernetes cluster.
2. Have access to an object store of your choice (for example, S3, **Glasglow Coma Scale** (**GCS**), or Minio). This object store will serve as the storage backing for the Pachyderm cluster where all the versioning and processed data is stored.
3. Deploy Pachyderm on the Kubernetes cluster using the Pachyderm CLI `pachctl`.



Detailed instructions to deploy Pachyderm on any cloud provider or on-premise can be found at `http://pachyderm.readthedocs.io/en/latest/deployment/deploy_intro.html`. Alternatively, you can easily experiment with and develop against a local Pachyderm cluster using `minikube`, as further detailed at `http://pachyderm.readthedocs.io/en/latest/getting_started/local_installation.html`.

Following one set of those instructions should get you to a state where you have a Kubernetes cluster running in the following state (which can be checked using Kubernetes CLI tool called `kubectl`):

```
$ kubectl get all
NAME READY STATUS RESTARTS AGE
po/etcd-2142892294-38ptw 1/1 Running 0 2m
po/pachd-776177201-04l6w 1/1 Running 0 2m

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
svc/etcd 10.0.0.141 <nodes> 2379:32379/TCP 2m
svc/kubernetes 10.0.0.1 <none> 443/TCP 3m
svc/pachd 10.0.0.178 <nodes> 650:30650/TCP,651:30651/TCP,652:30652/TCP 2m

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deploy/etcd 1 1 1 1 2m
deploy/pachd 1 1 1 1 2m
```

```
NAME DESIRED CURRENT READY AGE
rs/etcd-2142892294 1 1 1 2m
rs/pachd-776177201 1 1 1 2m
```

Here, you can see that the Pachyderm daemon (or server), `pachd`, is running in the Kubernetes cluster as a **pod**, which is simply a group of one or more containers. Our pipeline stages will also run as Kubernetes pods, but you won't have to worry too much about that as Pachyderm will take care of the details.



The ports and IPs listed in the above output may vary depending on your deployment and various configurations. However, a healthy Pachyderm cluster should look very similar.

If your Pachyderm cluster is running and you followed one of the Pachyderm deploy guides, you should also have the `pachctl` CLI tool installed. When `pachctl` is successfully connected to the Pachyderm cluster, you can run the following as a further sanity check (where the version number may change based on the Pachyderm version you are running):

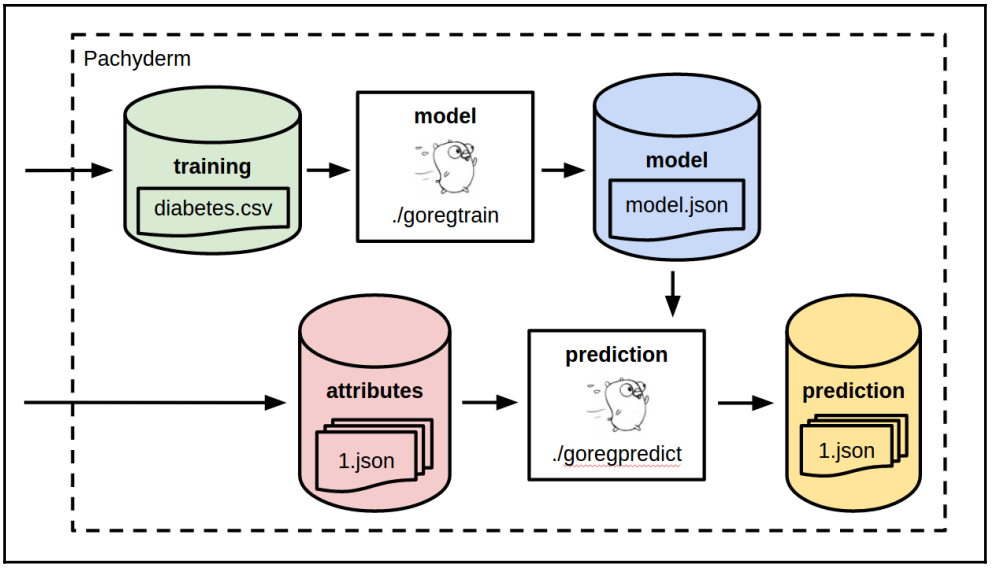
```
$ pachctl version
COMPONENT           VERSION
pachctl             1.6.0
pachd               1.6.0
```



If you have any trouble getting Pachyderm deployed or have any trouble building or running pipelines, the Pachyderm community has a great public Pachyderm Slack channel. You can join by visiting `http://slack.pachyderm.io/`. The community is active there every day, and any questions you have will be welcome.

Building a Pachyderm machine learning pipeline

As you have already seen, our example machine learning pipeline has two stages. The model stage of our pipeline will train a model and export a persisted version of this model to a file. The prediction stage of our pipeline will utilize the trained model to make predictions for input attribute files. Overall, this pipeline will look like the following in Pachyderm:



Each of the cylinders in the preceding figure represents a Pachyderm data repository of versioned data. You were already exposed to these data repositories in `Chapter 1`, *Gathering and Organizing Data*. Each of the boxes represents a containerized data pipeline stage. The basic units of data processing in Pachyderm are Docker images. Thus, we can utilize the Docker images that we created in preceding sections in this data pipeline.

By stringing together versioned collections of data (again, think about these as a sort of *Git for data*), processing these collections of data with containers, and saving results to other versioned collections of data, Pachyderm pipelines have some pretty interesting and useful implications.

First, we can go back at any point in time to see the state of any portion of our data at that point in time. This might help us as we further develop our model if we are wanting to develop a certain state of our data. It might also help us to collaboratively develop as a team by tracking the team's data over time. Not all changes to data are good, and we need the ability to revert after bad or corrupt data is committed into the system.

Next, all of our pipeline results are linked to the exact Docker images and exact states of other data that produced these results. The Pachyderm team calls this **provenance**. It is the ability to understand what pieces of processing and which pieces of data contributed to a particular result. For example, with this pipeline, we are able to determine the exact version of our persisted model that produced a particular result, along with the exact Docker image that produced that model and the exact state of the training data that was input to that exact Docker image. We are, thus, able to exactly reproduce an entire run of any pipeline, debug strange model behavior, and attribute results to the corresponding input data.

Finally, because Pachyderm knows which portions of your data repositories are new or different, our data pipeline can process data incrementally and can be triggered automatically. Let's say that we have already committed one million attribute files into the `attributes` data repository, and then we commit ten more attribute files. We don't have to reprocess the first million to update our results. Pachyderm knows that we only need to process the ten new files and will send these to the `prediction` stage to be processed. Moreover, Pachyderm will (by default) automatically trigger this update because it knows that the update is needed to keep the results in sync with the input.



Incremental processing and automatic triggers are some of the default behaviors of Pachyderm pipelines, but they are not the only things you can do with Pachyderm. In fact, we will only scratch the surface here of what is possible with Pachyderm. You can do distributed map/reduce style operations, distributed image processing, periodic processing of database tables, and much, much more. Thus, the techniques here should be transferable to a variety of domains.

Creating and filling the input repositories

To create the Pachyderm pipeline on our deployed Pachyderm cluster, we first need to create the input repositories for the pipeline. Remember, our pipeline has the `training` and `attributes` data repositories that drive the rest of the pipeline. As we put data into these, Pachyderm will trigger the downstream portions of the pipeline and calculate the results.

We saw in `Chapter 1`, *Gathering and Organizing Data*, how we can create data repositories in Pachyderm and commit data into these repositories using `pachctl`, but let's try to do this directly from a Go program here. In this particular example, there is not any real advantage to doing this, due to the fact that we already have our training and test data in files and we can easily commit those files into Pachyderm by name.

However, imagine a more realistic scenario. We might want our data pipeline to be integrated with some other Go services that we have already written at our company. One of these services might process incoming patient medical attributes from doctors or clinics, and we want to feed these attributes to our data pipeline, such that the data pipeline can make the corresponding predictions of disease progression.

In such a scenario, we would ideally like to pass the attributes directly into Pachyderm from our service instead of manually copying all that data. This is where the Pachyderm Go client, `github.com/pachyderm/pachyderm/src/client`, can come in very handy. Using the Pachyderm Go client, we can create our input repositories as follows (after connecting to the Pachyderm cluster):

```
// Connect to Pachyderm using the IP of our
// Kubernetes cluster. Here we will use localhost
// to mimic the scenario when you have k8s running
// locally and/or you are forwarding the Pachyderm
// port to your localhost.. By default
// Pachyderm will be exposed on port 30650.
c, err := client.NewFromAddress("0.0.0.0:30650")
if err != nil {
    log.Fatal(err)
}
defer c.Close()

// Create a data repository called "training."
if err := c.CreateRepo("training"); err != nil {
    log.Fatal(err)
}

// Create a data repository called "attributes."
if err := c.CreateRepo("attributes"); err != nil {
    log.Fatal(err)
}

// Now, we will list all the current data repositories
// on the Pachyderm cluster as a sanity check. We
// should now have two data repositories.
repos, err := c.ListRepo(nil)
if err != nil {
    log.Fatal(err)
}

// Check that the number of repos is what we expect.
if len(repos) != 2 {
    log.Fatal("Unexpected number of data repositories")
}
```

```
// Check that the name of the repo is what we expect.
if repos[0].Repo.Name != "attributes" || repos[1].Repo.Name != "training" {
    log.Fatal("Unexpected data repository name")
}
```

Compiling this and running it creates the `repos` as expected, which can be confirmed as follows:

```
$ go build
$ ./myprogram
$ pachctl list-repo
NAME CREATED SIZE
attributes 3 seconds ago 0B
training 3 seconds ago 0B
```



For simplicity, the above code just creates and checks the repo once. If you ran the code again, you would get an error because the repo was already created. To improve on this, you could check if the repo exists and then create it if it does not exist. You can also check that the repo exists by name.

Now, let's go ahead and put an attributes file in the `attributes` repository and the `diabetes.csv` training data set into the `training` repository. This is also very easily done directly in Go, via the Pachyderm client:

```
// Connect to Pachyderm.
c, err := client.NewFromAddress("0.0.0.0:30650")
if err != nil {
    log.Fatal(err)
}
defer c.Close()

// Start a commit in our "attributes" data repo on the "master" branch.
commit, err := c.StartCommit("attributes", "master")
if err != nil {
    log.Fatal(err)
}

// Open one of the attributes JSON files.
f, err := os.Open("1.json")
if err != nil {
    log.Fatal(err)
}

// Put a file containing the attributes into the data repository.
if _, err := c.PutFile("attributes", commit.ID, "1.json", f); err != nil {
    log.Fatal(err)
```

```
}

// Finish the commit.
if err := c.FinishCommit("attributes", commit.ID); err != nil {
    log.Fatal(err)
}

// Start a commit in our "training" data repo on the "master" branch.
commit, err = c.StartCommit("training", "master")
if err != nil {
    log.Fatal(err)
}

// Open up the training data set.
f, err = os.Open("diabetes.csv")
if err != nil {
    log.Fatal(err)
}

// Put a file containing the training data set into the data repository.
if _, err := c.PutFile("training", commit.ID, "diabetes.csv", f); err !=
nil {
    log.Fatal(err)
}

// Finish the commit.
if err := c.FinishCommit("training", commit.ID); err != nil {
    log.Fatal(err)
}
```

Running this does indeed commit the data into Pachyderm in the proper data repositories, as can be seen here:

```
$ go build
$ ./myprogram
$ pachctl list-repo
NAME CREATED SIZE
training 13 minutes ago 73.74KiB
attributes 13 minutes ago 210B
$ pachctl list-file training master
NAME TYPE SIZE
diabetes.csv file 73.74KiB
$ pachctl list-file attributes master
NAME TYPE SIZE
1.json file 210B
```



It is important to finish the commits that we started above. If you leave an orphaned commit hanging you might block other commits of data. Thus, you might consider deferring the finish of the commit, just to be safe.

Good! Now we have data in the proper input data repositories on our remote Pachyderm cluster. Next, we can actually process this data and produce results.

Creating and running the processing stages

Processing stages are created declaratively via a **pipeline specification** in Pachyderm. If you have worked with Kubernetes before, this type of interaction should be familiar. Basically, we declare to Pachyderm what processing we want to take place, and Pachyderm handles all the details to make sure that this processing happens as declared.

Let's first create the `model` stage of our pipeline using a JSON pipeline specification that is stored in `train.json`. This JSON specification is as follows:

```
{
  "pipeline": {
    "name": "model"
  },
  "transform": {
    "image": "dwhitena/goregtrain:single",
    "cmd": [
      "/goregtrain",
      "-inDir=/pfs/training",
      "-outDir=/pfs/out"
    ]
  },
  "parallelism_spec": {
    "constant": "1"
  },
  "input": {
    "atom": {
      "repo": "training",
      "glob": "/"
    }
  }
}
```

This might look somewhat complicated, but there are really only a few things happening here. Let's dissect the specification to learn about the different pieces:

1. First, we are telling Pachyderm to name our pipeline `model`.

2. Next, we are telling Pachyderm that we want this `model` pipeline to process data using our `dwhitena/goregtrain:single`, and we want the pipeline to run our `goregtrain` binary in the container when processing the data.
3. Finally, the specification tells Pachyderm that we intend to process the `training` data repository as input.

To understand some of the other details of the specification, we need to first discuss what will happen when we create the pipeline on our Pachyderm cluster using the specification. When this happens, Pachyderm will use Kubernetes to schedule one or more worker pods on the Kubernetes cluster. These worker pods will then wait, ready to process any data supplied to them by the Pachyderm daemon. When there is data that needs to be processed in the input repository, `training`, the Pachyderm daemon will trigger a job in which these workers will process the data. Pachyderm will automatically mount the input data in the container at `/pfs/<input repo name>` (`/pfs/training` in this case). Data the container writes out to `/pfs/out` will be automatically versioned in an output data repository with the same name as the pipeline (`model` in this case).

The `parallelism_spec` portion of the specification lets us tell Pachyderm how many workers it should spin up to process the input data. Here we will just spin up a single worker and process the data serially. Later in the chapter, we will discuss parallelizing our pipeline and the associated data sharding, which is controlled by the `glob` pattern in the `input` section of the specification.

Enough talk though! Let's actually create this `model` pipeline stage on our Pachyderm cluster and get to processing some data. The pipeline can easily be created as follows:

```
$ pachctl create-pipeline -f model.json
```

When we do this, we can see the worker(s) that Pachyderm creates on the Kubernetes cluster using `kubectl` as shown in the following code:

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
etcd-2142892294-38ptw 1/1 Running 0 2h
pachd-776177201-04l6w 1/1 Running 0 2h
pipeline-model-v1-p0lnf 2/2 Running 0 1m
```

We will also see that Pachyderm has automatically triggered a job to perform our model training. Remember, this happens because Pachyderm knows that there is data in the input repository, `training`, that has not been processed. You can see the triggered job along with its results as follows:

```
$ pachctl list-job
```

```
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
3 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
$ pachctl list-repo
NAME CREATED SIZE
model 3 minutes ago 160B
training About an hour ago 73.74KiB
attributes About an hour ago 210B
$ pachctl list-file model master
NAME TYPE SIZE k8s
model.json file 160B
$ pachctl get-file model master model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

Great! We can see that our model training produced the results that we expected. However, Pachyderm was smart enough to trigger the pipeline, and it will update this model whenever we modify the training dataset.

The `prediction` pipeline can be created in a similar manner. Its pipeline specification, `pipeline.json`, is shown here:

```
{
  "pipeline": {
    "name": "prediction"
  },
  "transform": {
    "image": "dwhitena/goregpredict",
    "cmd": [
      "/goregpredict",
      "-inModelDir=/pfs/model",
      "-inVarDir=/pfs/attributes",
      "-outDir=/pfs/out"
    ]
  },
  "parallelism_spec": {
    "constant": "1"
  },
  "input": {
    "cross": [
      {
```

```
          "atom": {
            "repo": "attributes",
            "glob": "/*"
          }
        },
        {
          "atom": {
            "repo": "model",
            "glob": "/"
          }
        }
      ]
    }
}
```

The major difference in this pipeline specification is that we have two input repositories rather than one. These two input repositories are the `attributes` repository that we previously created and the `model` repository that contains the output of the `model` pipeline. These are combined using a `cross` primitive. `cross` ensures that all relevant pairs of our model and attribute files are processed.



If you are interested, you can find out more about the other ways to combine data in Pachyderm in the Pachyderm docs; go to `http://pachyderm.readthedocs.io/en/latest/`.

Creating this pipeline and examining the results (corresponding to `1.json`) can be accomplished as follows:

```
$ pachctl create-pipeline -f prediction.json
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 About a minute ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
19 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
$ pachctl list-repo
NAME CREATED SIZE
prediction About a minute ago 266B
model 19 minutes ago 160B
training About an hour ago 73.74KiB
attributes About an hour ago 210B
$ pachctl list-file prediction master
NAME TYPE SIZE
1.json file 266B
$ pachctl get-file prediction master 1.json
```

```
{
    "predicted_diabetes_progression": 210.7100380636843,
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
        }
    ]
}
```

You can see that another data repository, `prediction`, was created to version the output of the `prediction` pipeline. In this manner, Pachyderm gradually builds up links between input data repositories, data processing stages, and output data repositories, such that it always knows what data and processing components are linked.

The full machine learning pipeline is now deployed and we have run each stage of the pipeline once. However, the pipeline is ready and waiting to process more data. All we have to do is put data at the top of the pipeline and Pachyderm will automatically trigger any downstream processing that needs to take place to update our results. Let's illustrate this by committing two more files into the `attributes` repository, as follows:

```
$ ls
2.json 3.json
$ pachctl put-file attributes master -c -r -f .
$ pachctl list-file attributes master
NAME TYPE SIZE
1.json file 210B
2.json file 211B
3.json file 211B
```

This will automatically trigger another prediction job to update our results. The new job can be seen in the following code:

```
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
be71b034-9333-4f75-b443-98d7bc5b48ab
prediction/bb2fd223df664e70ad14b14491109c0f About a minute ago Less than a
second 0 2 + 1 / 3 742B 536B success
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 8 minutes ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
26 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
```

Also, we can see that the `prediction` repository has two new results from running our prediction code again on the two new input files:

```
$ pachctl list-file prediction master
NAME TYPE SIZE
1.json file 266B
2.json file 268B
3.json file 268B
```



Note that Pachyderm did not reprocess `1.json` here even though it is shown with the latest results. Under the hood, Pachyderm knows that there was already a result corresponding to `1.json` in `attributes`, so there is no need to reprocess it.

Updating pipelines and examining provenance

Over time, we are likely going to want to update our machine learning models. The data that they are processing may have changed over time or maybe we have just developed a different or better model. In any event, it is likely that we are going to need to update our pipeline.

Let's assume that our pipeline is in the state described in the previous section, where we have created the full pipeline, the training process has run once, and we have processed two commits into the `attributes` repository. Now, let's update the model pipeline to train our multiple regression

Now, let's update the model pipeline to train our multiple regression models instead of our single regression model. This is actually really easy. We just need to change `"image": "dwhitena/goregtrain:single"` in our `model.json` pipeline specification to `"image": "dwhitena/goregtrain:multi"` and then update the pipeline as follows:

```
$ pachctl update-pipeline -f model.json --reprocess
```

When we give Pachyderm this new specification, Pachyderm will automatically update the worker pods such that they are running the containers with the multiple regression model. Also, because we supplied the `--reprocess` flag, Pachyderm will trigger one or more new jobs to reprocess any input commits with the new image that was previously processed with the old image. We can see these reprocessing jobs as follows:

```
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
03667301-980b-484b-afbe-7ea30da695f5
prediction/ef32a74b04ee4edda7bc2a2b469f3e03 2 minutes ago Less than a
```

```
second 0 3 + 0 / 3 1.355KiB 803B success
5587e13c-4854-4f3a-bc4c-ae88c65c007f model/f54ca1a0142542579c1543e41f5e9403
2 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 252B success
be71b034-9333-4f75-b443-98d7bc5b48ab
prediction/bb2fd223df664e70ad14b14491109c0f 16 minutes ago Less than a
second 0 2 + 1 / 3 742B 536B success
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 23 minutes ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
41 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
```

Looking at this output, we can notice something interesting and very useful. When our `model` stage trained our new multiple regression model and output this model as `model.json`, Pachyderm saw that the results of our `prediction` pipeline were now out of sync with the latest model. As such, Pachyderm automatically triggered another job to update our previous results using the new multiple regression models.

This workflow is super useful to manage both deployed models and during the development of models, because you do not need to manually shuffle around a bunch of model versions and related data. Everything happens automatically. However, a natural question arises from this update. How are we to know which models produced which results?

Well, this is where Pachyderm's data versioning combined with the pipelining really shines. We can look at any prediction result and inspect that commit to determine the commit's provenance. This is illustrated here:

```
$ pachctl inspect-commit prediction ef32a74b04ee4edda7bc2a2b469f3e03
Commit: prediction/ef32a74b04ee4edda7bc2a2b469f3e03
Parent: bb2fd223df664e70ad14b14491109c0f
Started: 8 minutes ago
Finished: 8 minutes ago
Size: 803B
Provenance: training/e0f9357455234d8bb68540af1e8dde81
attributes/2e5fef211f14498ab34c0520727296bb
model/f54ca1a0142542579c1543e41f5e9403
```

In this case, we can see that prediction `ef32a74b04ee4edda7bc2a2b469f3e03` resulted from model `f54ca1a0142542579c1543e41f5e9403`, which can be retrieved here:

```
$ pachctl get-file model f54ca1a0142542579c1543e41f5e9403 model.json
{
    "intercept": 152.13348416289583,
    "coefficients": [
        {
```

```
            "name": "bmi",
            "coefficient": 675.069774431606
        },
        {
            "name": "ltg",
            "coefficient": 614.9505047824742
        }
    ]
}
```

However, if we look at a previous prediction, we will see that it resulted from a different model:

```
$ pachctl inspect-commit prediction bb2fd223df664e70ad14b14491109c0f
Commit: prediction/bb2fd223df664e70ad14b14491109c0f
Parent: 5ce47c9e788d4893ae00c7ee6b1e8431
Started: 25 minutes ago
Finished: 25 minutes ago
Size: 802B
Provenance: model/a2c7b7dfb44a40e79318c2de30c7a0c8
training/e0f9357455234d8bb68540af1e8dde81
attributes/2e5fef211f14498ab34c0520727296bb
$ pachctl get-file model a2c7b7dfb44a40e79318c2de30c7a0c8 model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

We can always trace back the provenance of any piece of data in our pipeline, no matter how many jobs are run and how many times we update our pipeline. This is very important in creating sustainable machine learning workflows that can be maintained over time, improved upon, and updated without major headaches.

Scaling pipeline stages

Each pipeline specification in Pachyderm can have a corresponding `parallelism_spec` field. This field, along with the `glob` patterns in your inputs, lets you parallelize your pipeline stages over their input data. Each pipeline stage is individually scalable independent of all the other pipeline stages.

The `parallelism_spec` field in the pipeline specification lets you control how many workers Pachyderm will spin up to process data in that pipeline stage. For example, the following `parallelism_spec` would tell Pachyderm to spin up 10 workers for a pipeline:

```
"parallelism_spec": {
  "constant": "10"
},
```

The `glob` patterns in your inputs tell Pachyderm how it can share your input data over the workers declared by the `parallelism_spec`. For example, a `glob` pattern of `/*` would tell Pachyderm that it can split up your input data into datums consisting of any files of directories at the root of your repository. A glob pattern of `/*/*` would tell Pachyderm that it can split up your data into datums consisting of files or directories at a level deep into your repository directory structure. If you are new to glob patterns, check out this description with some examples: `https://en.wikipedia.org/wiki/Glob_(programming)`.

In the case of our example pipeline, we could imagine that we need to scale the `prediction` stage of our pipeline because we are seeing a huge influx of attributes that need corresponding predictions. If this was the case, we could change `parallelism_spec` in our `prediction.json` pipeline specification to `"constant": "10"`. As soon as we update the pipeline with the new specification, Pachyderm will spin up 10 new workers for the `prediction` pipeline, as follows:

```
$ pachctl update-pipeline -f prediction.json
 $ kubectl get pods
 NAME READY STATUS RESTARTS AGE
 etcd-2142892294-38ptw 1/1 Running 0 3h
 pachd-776177201-04l6w 1/1 Running 0 3h
 pipeline-model-v2-168k5 2/2 Running 0 29m
 pipeline-prediction-v2-0p6zs 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-3fbsc 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-c3m4f 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-d11b9 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-hjdll 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-hnk64 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-q29f1 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-qlhm4 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-qrnf9 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-rdt81 0/2 Init:0/1 0 6s
 $ kubectl get pods
 NAME READY STATUS RESTARTS AGE
 etcd-2142892294-38ptw 1/1 Running 0 3h
 pachd-776177201-04l6w 1/1 Running 0 3h
 pipeline-model-v2-168k5 2/2 Running 0 30m
 pipeline-prediction-v2-0p6zs 2/2 Running 0 26s
```

```
pipeline-prediction-v2-3fbsc 2/2 Running 0 26s
pipeline-prediction-v2-c3m4f 2/2 Running 0 26s
pipeline-prediction-v2-d11b9 2/2 Running 0 26s
pipeline-prediction-v2-hjdll 2/2 Running 0 26s
pipeline-prediction-v2-hnk64 2/2 Running 0 26s
pipeline-prediction-v2-q29f1 2/2 Running 0 26s
pipeline-prediction-v2-qlhm4 2/2 Running 0 26s
pipeline-prediction-v2-qrnf9 2/2 Running 0 26s
pipeline-prediction-v2-rdt81 2/2 Running 0 26s
```

Now, any new commits of attributes into the `attributes` repo will be processed in parallel by the 10 workers. For example, if we committed 100 more attribute files into `attributes`, Pachyderm would send 1/10 of those files to each of the 10 workers to process in parallel. All of the results would still be seen the same way in the `prediction` repo.

Of course, setting a constant number of workers is not the only way you can do scaling with Pachyderm and Kubernetes. Pachyderm will let you scale down workers as well by setting a scale down threshold for idle workers. In addition, Pachyderm auto-scaling of workers can be combined with auto-scaling of Kubernetes cluster resources to deal with bursts of data and batch processing. Still, further, Pachyderm will allow you to specify the resources needed for pipelines and, using this resource specification, you can offload machine learning workloads to accelerators (GPUs, for example).

References

Docker:

- Introduction to Docker: `https://training.docker.com/introduction-to-docker`
- Building minimal Docker images for Go apps: `https://blog.codeship.com/building-minimal-docker-containers-for-go-applications/`

Pachyderm:

- Public Slack channel: `http://slack.pachyderm.io/`
- Getting started guide: `http://pachyderm.readthedocs.io/en/latest/getting_started/getting_started.html`
- Go client docs: `https://godoc.org/github.com/pachyderm/pachyderm/src/client`

Machine learning with Pachyderm: `http://pachyderm.readthedocs.io/en/latest/cookbook/ml.html`

- Machine learning examples: `http://pachyderm.readthedocs.io/en/latest/examples/readme.html#machine-learning`
- Distributed processing with Pachyderm: `http://pachyderm.readthedocs.io/en/latest/fundamentals/distributed_computing.html`
- Common Pachyderm deployments: `http://pachyderm.readthedocs.io/en/latest/deployment/deploy_intro.html`
- Auto-scaling Pachyderm clusters: `http://pachyderm.readthedocs.io/en/latest/managing_pachyderm/autoscaling.html`

Summary

From merely gathering CSV data to a fully deployed, distributed machine learning pipeline, we did it! We can now build machine learning models with Go and deploy these models with confidence on a cluster of machines. These patterns should allow you to build and deploy a whole variety of intelligent applications, and I can't wait to hear about what you create! Please stay in touch.

10
Algorithms/Techniques Related to Machine Learning

There are a few algorithms and techniques, related to the machine learning examples in this book, that we were not able to go into much detail about in the preceding chapters. We will address that here.

Gradient descent

In multiple examples (including those in `Chapter 4`, *Regression* and `Chapter 5`, *Classification*), we took advantage of an optimization technique called **gradient descent**. There are multiple variants of the gradient descent method, and, in general, you will see them pretty much everywhere in the machine learning world. Most prominently, they are utilized in the determination of optimal coefficients for algorithms such as linear or logistic regression, and thus, they often also play a role in more complicated techniques at least partially based on linear/logistic regression (such as neural networks).

The general idea of gradient descent methods is to determine a direction and magnitude of change in some parameters that will move you in the right direction to optimize some measure (such as error). Think about standing on some landscape. To move toward lower elevations, you need to take steps in the downward direction. This is basically what gradient descent is doing algorithmically when it is optimizing parameters.

Let's gain some more intuition about this process by looking at so-called **Stochastic Gradient Descent** (**SGD**), which is an incremental kind of gradient descent. If you remember, we actually utilized SGD in our implementation of logistic regression in `Chapter 5`, *Classification*. In that example, we implemented the training or fitting of our logistic regression parameters as follows:

```
// logisticRegression fits a logistic regression model
// for the given data.
func logisticRegression(features *mat64.Dense, labels []float64, numSteps
int, learningRate float64) []float64 {

    // Initialize random weights.
    _, numWeights := features.Dims()
    weights := make([]float64, numWeights)

    s := rand.NewSource(time.Now().UnixNano())
    r := rand.New(s)

    for idx, _ := range weights {
        weights[idx] = r.Float64()
    }

    // Iteratively optimize the weights.
    for i := 0; i < numSteps; i++ {

        // Initialize a variable to accumulate error for this iteration.
        var sumError float64

        // Make predictions for each label and accumulate error.
        for idx, label := range labels {

            // Get the features corresponding to this label.
            featureRow := mat64.Row(nil, idx, features)

            // Calculate the error for this iteration's weights.
            pred := logistic(featureRow[0]*weights[0] +
featureRow[1]*weights[1])
            predError := label - pred
            sumError += math.Pow(predError, 2)

            // Update the feature weights.
            for j := 0; j < len(featureRow); j++ {
                weights[j] += learningRate * predError * pred * (1 - pred)
* featureRow[j]
            }
        }
   }
```

```
    return weights
}
```

The loop under the `// Iteratively optimize the weights` comment implements SGD to optimize the logistic regression parameters. Let's pick apart this loop to determine what exactly is happening.

First, we calculate the output of our model with the current weights and the difference between our prediction and the ideal value (the actual observation, that is):

```
// Calculate the error for this iteration's weights.
pred := logistic(featureRow[0]*weights[0] + featureRow[1]*weights[1])
predError := label - pred
sumError += math.Pow(predError, 2)
```

Then, according to SGD, we are going to calculate an update to our parameters (in this case `weights`) according to the following:

$$update = learning\ rate \times gradient\ of\ the\ parameters$$

The **gradient** is the mathematical gradient of the cost function in your problem.



More detailed mathematical information about this quantity can be found here:
`http://mathworld.wolfram.com/Gradient.html`

The update can then be applied to the parameters as follows:

$$parameters = parameters - update$$

In the case of our logistic regression model, this works out to have the following form:

```
// Update the feature weights.
for j := 0; j < len(featureRow); j++ {
    weights[j] += learningRate * predError * pred * (1 - pred) *
featureRow[j]
}
```

This type of SGD is pretty widely used in machine learning. However, in some cases, this kind of gradient descent can lead to overfitting or getting stuck in local minimums/maximums (rather than finding the global optimum).

To address some of these issues, you can utilize a variant of gradient descent called **batch gradient descent**. In batch gradient descent, you calculate each of the parameter updates based on gradients in all of the training dataset, as opposed to a gradient for a particular observation or row of the dataset. This helps you prevent overfitting, but it can also be rather slow and have memory issues because you need to calculate gradient with respect to a whole dataset for each parameter. **Mini-batch gradient descent**, which is another variant, attempts to maintain some of the benefits of batch gradient descent while being more computationally tractable. In mini-batch gradient descent, the gradients are calculated on subsets of the training dataset rather than the whole training dataset.



In the case of logistic regression, you may see the use of gradient ascent or descent, where gradient ascent is the same thing as gradient descent except that it is applied to the negative of the cost function. The logistic cost function gives you both of these options as long as you are consistent. This is further discussed at `https://stats.stackexchange.com/questions/261573/using-gradient-ascent-instead-of-gradient-descent-for-logistic-regression`.



Gradient descent methods are also already implemented by the gonum team in `gonum.org/v1/gonum/optimize`. See these docs for more information:
`https://godoc.org/gonum.org/v1/gonum/optimize#GradientDescent`

Entropy, information gain, and related methods

In `Chapter 5`, *Classification*, we explored decision tree methods in which models consisted of a tree of if/then statements. These if/then portions of the decision tree split the prediction logic based on one of the features of the training set. In an example where we were trying to classify medical patients into unhealthy or healthy categories, a decision tree might first split based on a gender feature, then based on an age feature, then based on a weight feature, and so on, eventually landing on healthy or unhealthy.

How does the algorithm choose which features to use first in the decision tree? In the preceding example, we could split on gender first, or weight first, and any other feature first. We need a way to arrange our splits in an optimal way, such that our model makes the best predictions that it can make.

Many decision tree model implementations, including the one we used in `Chapter 5`, *Classification*, use a quantity called **entropy** and an analysis of **information gain** to build up decision trees. To illustrate this process, let's consider an example. Assume that you have the following data about numbers of healthy people versus various characteristics of those people:

	Healthy	**Unhealthy**
Vegan Diet	5	2
Vegetarian Diet	4	1
Meat Eating Diet	3	4

	Healthy	**Unhealthy**
Age 40+	3	5
Age < 40	9	2

Here, we have two features in our data, Diet and Age, and we would like to build a decision tree to predict if people are healthy or not based on Diet and Age. To do this, we need to decide whether we should split our decision tree first on Age or first on Diet. Notice that we also have a total of 12 healthy people and 7 unhealthy people represented in the data.

To begin, we will calculate the overall or total entropy of the classes in our data. This is defined as follows:

$$E = -p_1 log_2(p_1) - p_2 log_2(p_2) - p_3 log_2(p_3) - \ldots$$

Here, p_1, p_2, and so on, are the probabilities of a first category, a second category, and so on. In our particular case (because we have 12 healthy people and 7 unhealthy people), our total entropy is as follows:

$$E(Health) = -\left(\frac{12}{19}\right) log_2\left(\frac{12}{19}\right) - \left(\frac{7}{19}\right) log_2\left(\frac{7}{19}\right) = 0.95$$

This *0.95* measure represents the homogeneity of our health data. It goes between 0 and 1, with high values corresponding to less homogeneous data.

To determine whether we should split our tree first on Age or first on Diet, we will calculate which of these features gives us the most information gain. Simply put, we will find the feature that gives us the most homogeneity after splitting on that feature, as measured by the preceding entropy. This decrease in entropy is called **information gain**.

The information gain for a certain feature in our example is defined as follows:

$$G = E(Health) - E(Health, Feature)$$

Here, *E(Health, Feature)* is a second measure of entropy with respect to the given feature (*Age* or *Diet*). For Diet, this second measure would be calculated as follows:

$$E(Health, Age) = p_{40+}E(Health, 40+) + p_{<40}E(Health, < 40)$$

The quantities p_{40+} and $p_{<40}$ are the probabilities of having an age of *40+* or *<40* (8/19 and 11/19, respectively). The quantities *E(Health,40+)* and *E(Health,<40)* are the health entropies (as defined in the preceding formula) but only using the counts corresponding to those Age *40+* and Age *<40*, respectively.

For our example data, the information gain for the Age feature comes out to *0.152* and the information gain for the Diet feature comes out to *0.079*. Thus, we would choose to split our decision tree on the Age feature first because it increases the overall homogeneity of our data the most.



You can find out more about building decision trees based on entropy at `http://www.saedsayad.com/decision_tree.htm`, and you can see an example implementation in Go at `https://github.com/sjwhitworth/golearn/blob/master/trees/entropy.go`.

Backpropagation

`Chapter 8`, *Neural Networks and Deep Learning*, included an example of a neural network built from scratch in Go. This neural network included an implementation of the backpropagation method to train neural networks, which can be found in almost any neural network code. We discussed some details in that chapter. However, this method is utilized so often that we wanted to go through it step by step here.

To train a neural network with backpropagation, we do the following for each of a series of epochs:

1. Feed the training data through the neural network to produce output.
2. Calculate an error between the expected output and the predicted output.
3. Based on the error, calculate updates for the neural network weights and biases.
4. Propagate these updates back into the network.

As a reminder, our implementation of this procedure for a network with a single hidden layer looked like the following (where `wHidden` and `wOut` are our hidden layer and output layer weights and `bHidden` and `bOut` are our hidden layer and output layer biases):

```
105     // Define the output of the neural network.
106     output := mat.NewDense(0, 0, nil)
107
108     // Loop over the number of epochs utilizing
109     // backpropagation to train our model.
110     for i := 0; i < nn.config.numEpochs; i++ {
111
112         // Complete the feed forward process.
113         hiddenLayerInput := mat.NewDense(0, 0, nil)
114         hiddenLayerInput.Mul(x, wHidden)
115         addBHidden := func(_, col int, v float64) float64 { return v +
bHidden.At(0, col) }
116         hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)
117
118         hiddenLayerActivations := mat.NewDense(0, 0, nil)
119         applySigmoid := func(_, _ int, v float64) float64 { return
sigmoid(v) }
120         hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)
121
122         outputLayerInput := mat.NewDense(0, 0, nil)
123         outputLayerInput.Mul(hiddenLayerActivations, wOut)
124         addBOut := func(_, col int, v float64) float64 { return v +
bOut.At(0, col) }
125         outputLayerInput.Apply(addBOut, outputLayerInput)
126         output.Apply(applySigmoid, outputLayerInput)
127
128         // Complete the backpropagation.
129         networkError := mat.NewDense(0, 0, nil)
130         networkError.Sub(y, output)
131
132         slopeOutputLayer := mat.NewDense(0, 0, nil)
133         applySigmoidPrime := func(_, _ int, v float64) float64 { return
sigmoidPrime(v) }
134         slopeOutputLayer.Apply(applySigmoidPrime, output)
```

```
135         slopeHiddenLayer := mat.NewDense(0, 0, nil)
136         slopeHiddenLayer.Apply(applySigmoidPrime,
hiddenLayerActivations)
137
138         dOutput := mat.NewDense(0, 0, nil)
139         dOutput.MulElem(networkError, slopeOutputLayer)
140         errorAtHiddenLayer := mat.NewDense(0, 0, nil)
141         errorAtHiddenLayer.Mul(dOutput, wOut.T())
142
143         dHiddenLayer := mat.NewDense(0, 0, nil)
144         dHiddenLayer.MulElem(errorAtHiddenLayer, slopeHiddenLayer)
145
146         // Adjust the parameters.
147         wOutAdj := mat.NewDense(0, 0, nil)
148         wOutAdj.Mul(hiddenLayerActivations.T(), dOutput)
149         wOutAdj.Scale(nn.config.learningRate, wOutAdj)
150         wOut.Add(wOut, wOutAdj)
151
152         bOutAdj, err := sumAlongAxis(0, dOutput)
153         if err != nil {
154             return err
155         }
156         bOutAdj.Scale(nn.config.learningRate, bOutAdj)
157         bOut.Add(bOut, bOutAdj)
158
159         wHiddenAdj := mat.NewDense(0, 0, nil)
160         wHiddenAdj.Mul(x.T(), dHiddenLayer)
161         wHiddenAdj.Scale(nn.config.learningRate, wHiddenAdj)
162         wHidden.Add(wHidden, wHiddenAdj)
163
164         bHiddenAdj, err := sumAlongAxis(0, dHiddenLayer)
165         if err != nil {
166             return err
167         }
168         bHiddenAdj.Scale(nn.config.learningRate, bHiddenAdj)
169         bHidden.Add(bHidden, bHiddenAdj)
170     }
```

Let's pick this implementation apart in detail to understand exactly what's happening.

The feed forward process that produces our output does the following:

1. Multiplies the input data by the hidden layer weights, adds the hidden layer biases, and applies the sigmoid activation function to calculate the output of the hidden layer, `hiddenLayerActivations` (lines 112 to 120 in the preceding snippet).

2. Multiples the `hiddenLayerActivations` by the output layer weights, then adds the output layer biases, and applies the sigmoid activation function to calculate the `output` (lines 122 to 126).

Notice that in the feed forward process, we are starting with the input data at the input layer and working our way forward through the hidden layer until we reach the output.

After the feed forward process, we need to calculate optimal updates to our weights and biases. As you might expect after going through the gradient descent portion of this Appendix, gradient descent is a perfect fit to find these weights and biases. Lines 129 through 144 in the preceding snippet implement SGD.

Finally, we need to apply these updates backward through the network in lines 147 through 169. This is the backward propagation of updates that gives backpropagation its name. There isn't anything too special about this process, we just perform the following:

1. Apply the calculated updates to the output weights and biases (lines 147 to 157).
2. Apply the calculated updates to the hidden layer weights and biases (lines 159 to 169).

Notice how we start at the output and work our way back to the input applying the changes.



You can find a very detailed discussion of backpropagation, including mathematical proofs, here:
`http://neuralnetworksanddeeplearning.com/chap2.html`



Backpropagation started to be widely utilized after a 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. Although the method is utilized across the industry in neural networks, Geoffrey Hinton recently came out to say that he is *deeply suspicious* of backpropagation and suggests that we need to work hard to find an alternative.

Index

A

accuracy 68
ACF plots
  reference 180
Adjusted Rand index 156
Amazon EC2 Container Registry (ECR) 233
Amazon Web Service (AWS) 233
anomaly detection 197, 199
area under the curve (AUC) 72
assumptions, linear regression
  homoscedasticity 89
  linear relationship 89
  no auto-correlation 89
  no multicollinearity 89
  normality 89
assumptions, multiple linear regression
  overfitting 105
  relative scale 105
auto-regressive (AR)
  for forecasting 184
auto-regressive integrated moving average (ARIMA) models 197
auto-regressive model example
  about 186
  ACF, analyzing 189
  AR order, selecting 189
  AR(2) model, evaluating 190
  AR(2) model, fitting 190
  stationary series 186, 188
auto-regressive model
  assumptions 185
  overview 184
auto-regressive moving average (ARMA) 197
autocorrelation 176, 177, 178
autocorrelation function (ACF) 176

B

backpropagation 208, 271
batch gradient descent 269
black boxes 203
box plot
  about 52
  generating 54

C

caching 27
categorical metrics 67
categorical variables
  individual evaluation metrics 67, 70
central tendency measures
  Mean 43
  Median 43
  Mode 43
centroids 151
chi-squared distribution 60
Citi Bike API
  reference 20
classification model 116
classification
  about 85
  concepts 116
clustering techniques
  about 169
  evaluating 151
clustering
  concepts 149
coefficient of determination 66
completeness 156
concepts, classification
  annotated data 116
  binary classification 116
  categories 116

classes 116
labeled data 116
labels 116
multi-class classification 116
concepts, clustering
clusters 149
distance 149
external criteria 149
groups 149
inter-cluster 149
inter-group 149
internal criteria 149
intra-cluster 149
intra-group 149
similarity 149
concepts, Docker
container 233
Docker engine 233
Docker image 233
Docker registry 233
Dockerfile 233
software container 233
concepts, neural net
activation 202
activation function 202
architecture 202
backpropagation 202
biases 202
feedforward 202
hidden layers 202
input layer 202
neurons 202
nodes 202
output layer 202
perceptrons 202
weights 202
concepts, time series
auto-regressive model 175
datetime 175
measurement 175
moving average model 175
observation 175
random variable 175
seasonality 175
stationary 175
time 175
time period 175
timestamp 175
trends 175
conditional probability 56
confusion matrices 72, 73
continuous metrics 63, 64, 66
convolutional layers 222
convolutional neural network (CNN) 222
cosine similarity 137
cross validation 81, 82
CSV data
manipulating, with data frames 18, 19
reading, from file 13, 14
unexpected fields, handling 14, 15
unexpected types, handling 15, 16
CSV files 12

D

data frames
CSV data, manipulating with 18, 19
data versioning
collaboration 30
compliance 30
creativity 30
data
best practices 11, 12
caching, in memory 28
handling, Gopher style 9, 10, 11
decision trees
about 140
example 142
overview 140
pitfalls 141
deep learning model
about 221
references 221
deep learning
about 220
references 220
with Go 222
degrees of freedom 60
differencing 186
distance
measuring 149

distributions
  about 42
  visualizing 48
Docker Hub 233
Docker-izing 233
Docker
  concepts 232
  reference 232, 233

E

elbow graph approach 160
entropy 140, 270
error 175
errors 88
Euclidean distance 150
evaluation 62
external clustering evaluation 156

F

False Negative (FN) 68
False Positive (FP) 68
file
  CSV data, reading from 13, 14
forecasting 171
Fowlkes-Mallows scores 156

G

Golang 150
Gopher 150
Gopher style
  data, handling 9, 10, 11
gradient descent 88, 266

H

Hamming distance 150
histograms
  about 48
  creating 49, 51
holdout set 80
Holt-Winters method 197
hypothesis testing
  about 57
  test statistics 58

I

independent probability 56
individual evaluation metrics
  for categorical variables 67, 70
InfluxDB
  reference 197
information gain 270, 271
instances 138
internal clustering
  evaluating 151, 153
Internet of Things (IoT) 171

J

Jaccard similarity 137
JSON
  about 20
  output 23
  parsing 20, 21, 22

K

k-means clustering example
  about 161
  clusters, generating 164
  data, profiling 161, 162, 163
  generated clusters, evaluating 166, 168
k-means clustering
  about 157
  assumptions 160
  overview 157, 158, 159
  pitfalls 160
k-nearest neighbors (kNN)
  about 135
  example 138
  overview 135, 136
  pitfalls 137
Kubernetes cluster
  setting up 248
Kubernetes
  reference 31

L

lasso regression 109
learning rate 211
linear regression example

about 90
data, profiling 90, 92, 93
independent variable, selecting 94, 96
model, training 99, 100
test set, creating 97, 98
trained model, evaluating 100, 102, 103, 104
training set, creating 97, 98
linear regression
about 86
assumptions 89
overview 87, 88
pitfalls 89
logistic function 117
logistic regression example
about 121
data, cleaning 122, 123, 125, 127
data, profiling 122, 123, 125
test set, creating 127, 129
training set, creating 127, 129
logistic regression model
testing 129, 131, 132
training 129, 131, 132
logistic regression
about 116
assumptions 121
overview 117, 119
pitfalls 121
logit 119
Long Short-Term Memory (LSTM) cells 222
Lossy Counting Algorithm (LCA) 198

M

machine learning application
Docker images, running on remote machines 246
Docker images, testing locally 244, 245
Docker-izing 234
model predictions, Docker-izing 239, 241, 242, 243
model training, Docker-izing 234, 235, 237, 238, 239
machine learning models
pipelines, updating 260
Manhattan distance 137, 150
matrices 35, 38, 39
matrix operations 40, 41
mean absolute error (MAE) 64
Mean Absolute Error (MAE) 192
mean squared deviation (MSD) 64
mean squared error (MSE) 64
memory
data, caching in 28
mini-batch gradient descent 269
Minkowski distance 137, 150
models
running, on remote machines 232
moving average models 197
multiple linear regression 104, 107, 108
assumptions 105
Mutual Information 156

N

Naive bayes
about 144
assumptions 144
example 145, 146
overview 144
neural network
architecture 206
concepts 202
evaluating 218, 219
training 208, 209, 210, 211, 213
training, on real data 216, 217
nodes
of neural network 204
non-linear regression 109

O

odds ratio 119
ordinary least squares (OLS) 87
overfitting
about 105
preventing 76

P

p-values
calculating 59
Pachyderm cluster
setting up 248
Pachyderm machine learning pipeline

building 250
input repositories, creating 251, 253
input repositories, filling 251, 253
processing stages, creating 255
processing stages, running 256
Pachyderm Slack channel
reference 32
Pachyderm
branches 31
commits 31
data repositories, creating for data versioning 32
data, inserting into data repositories 33
data, obtaining out of versioned data repositories 33
deploying 31
files 31
installing 31
reference 31
repositories 31
partial autocorrelation 181
partial autocorrelation function (PACF) 181
pipeline stages
scaling 262
precision 68
pretrained TensorFlow model
calling 224
retrieving 224
probability 55
probability measures 56
Prometheus
reference 197

Q

quantile-quantile (q-q) plot 94

R

R-squared 66
random forests
about 140
example 143
overview 141
pitfalls 141
random variables 55
recall 68
Receiver Operating Characteristic (ROC) curves 72
recurrent neural network 222
regression 85
regression model 85
regression, processes
attributes 86
dependent variables 85
explanatory variables 86
features 86
fitting 86
independent variables 86
linear regression 86
multiple regression 86
nonlinear regression 86
prediction 86
regressors 86
response variable 85
training 86
regularization 105
relative scale 105
remote machines
models, running on 232
reproduce 30
reproducible machine learning pipeline
building 247
residuals 88
ridge regression
about 109
implementing 109
ROC curve
generating 74

S

Sajari
about 99
reference 99
scalable machine learning pipeline
building 247
sigmoid function 205
silhouette coefficient 151
simple neural network
building 203
utilizing 215
skewed distribution 45
slices 36

spread measures
  maximum 45
  minimum 45
  quantiles/quartiles 45
  range 45
  standard deviation 45
  variance 45
SQL database
  connecting to 24, 25
  modifying 27
  querying 25, 26
SQL-like databases 24
squared Euclidean distance 150
statistical measures
  about 43
  central tendency measures 43
  dispersion measures 43, 45
  spread measures 45, 47
statistics 42
statistics, related to time series
  about 176
  autocorrelation 176, 177, 178, 180
  partial autocorrelation 181, 182, 184
stochastic gradient descent 129
Stochastic Gradient Descent (SGD) 267

T

TensorFlow
  about 223
  setting up 224
test set 76, 77, 78
test statistics 58
time series data
  representing 172, 173, 174
time series
  about 171
  concepts 175
training set 76, 77, 78
True Negative (TN) 68
True Positive (TP) 68

V

V-measures 156
validation 75
validation set 81
vector operations 36
vectors 36