The Machine Learning Handbooks

Volume I: Machine Learning Using R

Dr. KJG Mazidi

WWW.KARENMAZIDI.COM

THIRD EDITION, 2025

Created with a LaTeX template by Mathias Legrand `http://www.LaTeXTemplates.com`

Contents

Preface to the Third Edition

I accidentally started writing the first edition of this book during Spring Break 2018. I was driving down a long Texas road out in the country, thinking about a problem with the undergraduate machine learning class I was teaching. Chiefly, there was no book suitable for my course. There are several excellent machine learning books for graduate students, but most are too heavy on the math and too light on practical issues for undergraduate students. There are a few books for undergraduate students, but they didn't cover all the topics I was required to teach for my course at The University of Texas at Dallas. The solution I came up with was to format my own notes in LaTeX. I scribbled an outline on a piece of paper as I drove down the road. When I came home I started looking for a suitable template and found this one which I really liked. It was designed for a book and I thought, why not just make this a book? And so the book began.

I created the second edition of this book in 2020 to include both R and Python, since Python was becoming perhaps the most important language in machine learning. However, I always told my students that having both R and Python on their resumes would take them farther than having just one alone. That was the primary motivation for the second edition.

The first and second editions covered what I taught in a one-semester undergraduate Introduction to Machine Learning course. The third edition takes my readers a little further. For this third edition in 2025 I wanted to include PyTorch as well as TensorFlow. The book at that point would become massive, so I decided to break the book into a quartet:

- Volume I focuses on R and the many aspects of data exploration that are important skills for machine learning practitioners as well as data scientists; in addition, Volume I explores linear and statistical methods of machine learning
- Volume II focuses on Python and the sklearn platform for additional machine learning algorithms
- Volume III provides an introduction to TensorFlow and the Hugging Face platform
- Volume IV is all about PyTorch, which is rapidly becoming the most popular deep learning platform

The book has a companion Github site for code samples: `https://github.com/kjmazidi/Machine_Learning_3rd_edition/`

In addition, helpful tutorial videos on my YouTube channel continue to be added: `https://www.youtube.com/c/JaniceMazidi`

I've decided to make this 3rd edition freely available on the GitHub repo. I always gave the pdf to my students for free, so now I will consider anyone who wants to learn to be my student.

Finally, you can find me at `www.karenmazidi.com` or `www.mazidi-ed.com` to let me know your thoughts.

Happy coding!

Part I: Data Exploration with R

1. The Craft of Machine Learning

Volume I of *The Machine Learning Handbooks* provides an introduction to data exploration and machine learning with the R language. No prior experience with machine learning, or R, is assumed. The target audience is upper-level undergraduate or postgraduate students in computer science, as well as other disciplines, since machine learning has found application in a wide variety of fields from the social sciences, biological sciences, economics, and more. This volume will also be a good resource for professionals wanting to get started with machine learning. Prior courses in linear algebra, probability and statistics are nice to have; however, the book attempts to fill in as much detail as needed for the subject at hand.

The aim of this first volume of the series is to provide an introduction to the craft of machine learning through conceptual explanations of the algorithms and small examples of running the algorithms in R. Sample notebooks are provided on the author's github at: `https://github.com/kjmazidi/Machine_Learning_3rd_edition`.

1.1 Machine Learning

Figure 1.1 shows fields related to machine learning. Machine learning would not be possible without the fields surrounding it in the figure. Statistics and probability form the mathematical foundations of many of the algorithms described in this book. AI and computer science pushed the frontiers of what computers could do which made machine learning possible.

The term *machine learning* is an umbrella term for many closely related fields. Some statisticians call machine learning *statistical learning*. The field of *data science* and the task of *data mining* often involve machine learning techniques, coupled with more data exploration and analysis.

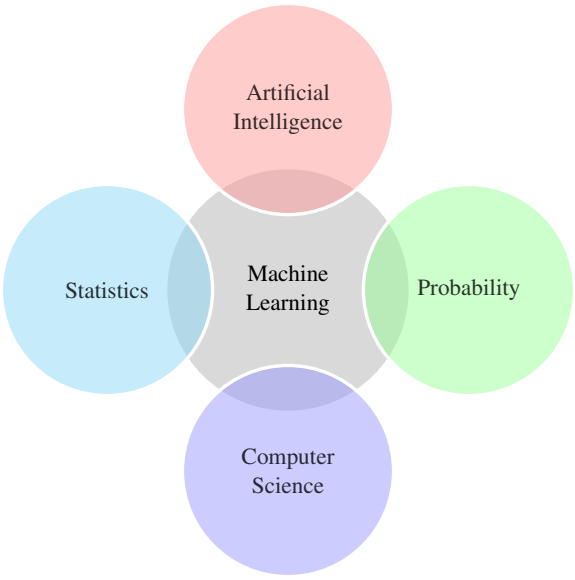


Figure 1.1: Fields Related to Machine Learning

Machine learning has received varied definitions as the field developed. This book proposes the following definition:

Definition 1.1.1 — Machine Learning. Machine learning trains computers to accurately recognize patterns in data for purposes of data analysis, prediction, and/or action selection by autonomous agents.

The key words in this definition are: data, patterns, predictions, and actions, along with the caveat: accurate. Let's examine these in detail.

1.1.1 Data

Nothing can be learned without data. The data and what we wish to learn from the data go hand in hand. For many learning scenarios, data takes the form of a table of values where each row represents one data example, and columns represent attributes or features of the examples. One learning scenario, clustering, seeks to group similar examples. Another learning scenario, supervised learning, seeks to learn about one feature based on combinations of the other features. Data can take other forms besides tables, such as a set of actions weighted by features that represent the current environment.

The data used in this handbook is small and neat compared to data you will encounter in real-world machine learning problems. This is by design, so that focus can be placed on the algorithms and how they learn from data. Just be aware that in real-world scenarios, more of your time will be spent in data gathering and data cleaning than in the actual machine learning.

When working with data, ethical considerations must always guide our actions. Who owns the data? Who are the subjects of the data? Has the data been anonymized? Did the subjects give consent for the use of the data? How will the analysis of this data be used? How might it impact the subjects in the data as well as the larger community?

1.1.2 Patterns

The best *general* pattern recognition machine is the human mind, but computers can in many cases beat human performance on narrowly defined tasks. The ability to recognize patterns in data enables algorithms to learn things like whether someone is a good credit risk, whether two people might be compatible, whether the object outlined in sensors is a human or a dark spot on the pavement.

When beginning a real-world machine learning project, how do you know what to look for? From raw data, organized data must be built. Once the data is organized, decisions must be made about what could be learned and what is important to learn from the data. These decisions often need to be in concert with domain experts and/or the owners and users of the data who wish to learn from it.

1.1.3 Predictions from Data

Learning patterns in data enables us to predict outcomes on future data – data the algorithm has never before seen. Predictions may simply involve finding similar instances in the data, or predicting a target value which may be quantitative or qualitative.

In the examples in this book, generally we train algorithms on a portion of the data and use the remaining data to test and evaluate how well the trained model can perform on previously unseen data. This is a common situation in machine learning, sometimes called batch learning because the data is fed into the algorithm in one batch. There are other approaches, however, such as online learning in which the algorithm is continually learning from newly available data and being evaluated in real time. Online learning techniques can also be used when the available data is too large to be stored in memory. An alternate approach to handle big data is to do parallel distributed machine learning, often in the cloud with specialized software.

1.1.4 Accuracy

Predictions must be accurate or they are not predictions but random guesses. Typically, predictions should beat some predetermined baseline approach. For example if 99% of the observations in a credit data set did not default on their loans, the goal is to beat a simple baseline that always guesses "not default" and has 99% accuracy. Machine learning makes use of many measurement techniques to gauge accuracy and evaluate performance of the algorithms. Many of these metrics are used to evaluate the training model itself and others will be used to evaluate performance on a held-out test set.

1.1.5 Actions

Every day more autonomous agents enter our lives, from smart thermostats, to automated assistants, to self-driving vehicles. These agents take actions based on what they have learned, and most continue to learn over time, usually by uploading data to a central learning repository. Some actions taken by autonomous agents will be controversial in the coming decades as ethical and legal issues evolve in response to humans co-existing with autonomous agents. The big players in AI are at the forefront of autonomous agents because they have the resources,

expertise, and data to pursue big projects. In the future, we expect the development of autonomous agents to be available to more developers.

Exercise 1.1 — Check your Understanding. One of the most difficult things to do when you start working with machine learning is to explain it to others. For this exercise, pretend you are talking to a friend who is in an unrelated field.

- How would you define *machine learning* in your own words?
- List 3 things to be aware of when collecting data for a machine learning project.
- What is the relationship between pattern recognition and machine learning?

1.2 Machine Learning Scenarios

There are scores of machine learning algorithms with countless variations each. This series will describe the most common algorithms, while providing a foundation for students to learn more algorithms on their own. There are many ways to classify machine learning algorithms, and not all algorithms fit neatly into categories but the following is a general overview.

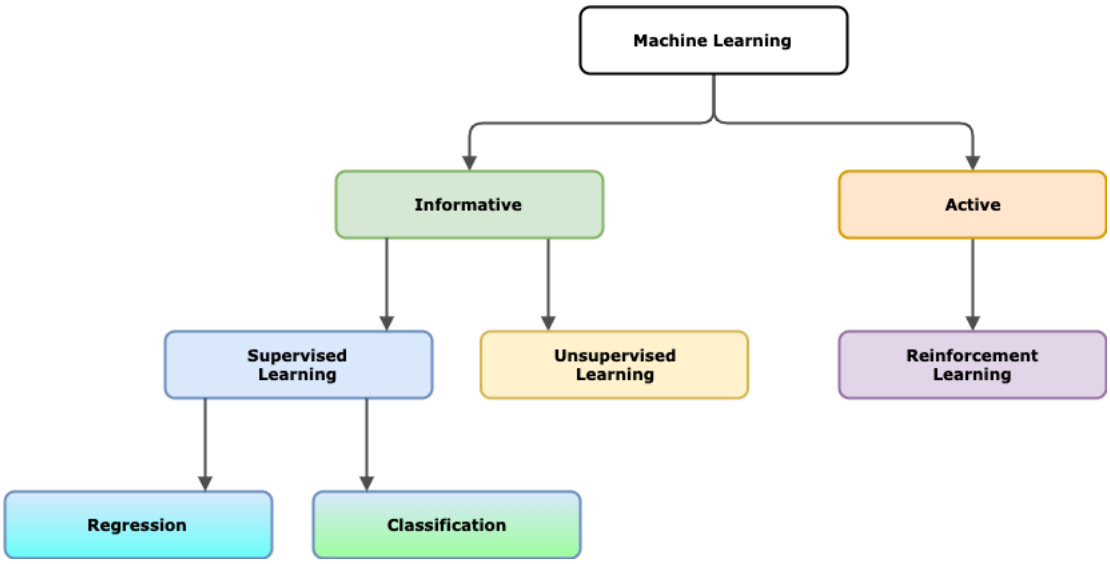


Figure 1.2: Machine Learning Scenarios

1.2.1 Informative v. Active

Most machine learning algorithms covered in this series are **informative** algorithms used for data analysis or prediction. These informative algorithms input data observations and output a model of the data that can then be used to predict outcomes for new data fed into the model. In contrast, the field of Reinforcement Learning teaches **active** agents to identify optimal actions given the current environment and what has been learned in past experience. The input to these algorithms for initial training comes in the form of data but some agents may continue to learn with sensors and other input methods that let them learn from the environment.

1.2.2 Supervised v. unsupervised learning

Informative algorithms are of two main types. The term **supervised learning** refers to scenarios where each data instance has a label. This label is used to train the algorithm so that labels can be predicted for future data items. The term **unsupervised learning** refers to scenarios where data does not have labels and the goal is simply to learn more about the data.

1.2.3 Regression v. classification

Supervised learning algorithms fall into two major groups. In **regression**, the target (or label) is a real-numbered **quantitative** value, like trying to predict the market value of a home given its square footage and other data. In **classification**, the target is **qualitative**, a class, like predicting if a borrower is a good credit risk or not, given their income, outstanding credit balance, and other predictors.

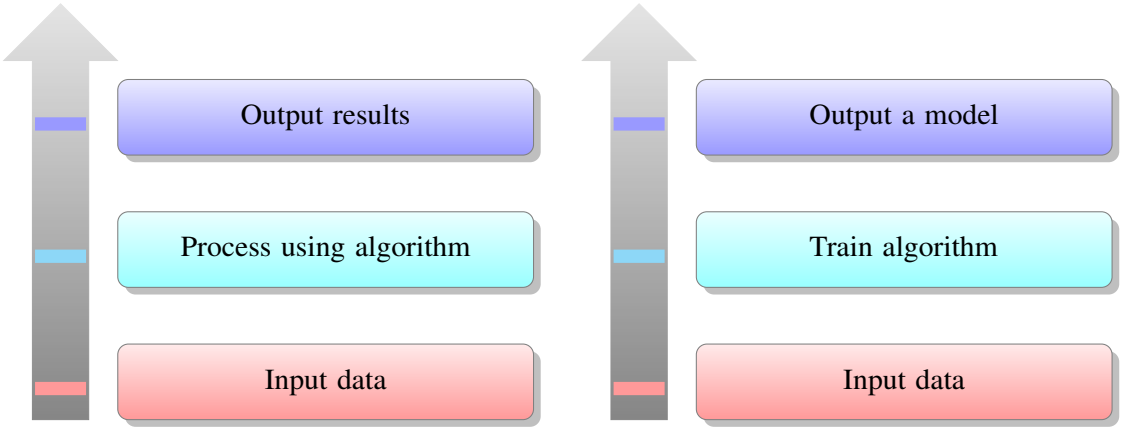


Figure 1.3: Traditional Programming (Left) v. Machine Learning (Right)

1.3 Machine learning v. traditional algorithms

Machine learning algorithms are different from traditional algorithms encountered in computer programming. Figure 1.3 shows the traditional computer programming paradigm on the left: data is fed into code that processes it and outputs the results of the processing. In the same figure, the machine learning paradigm is on the right: data is fed into an algorithm which builds and outputs a model of the data. In traditional programming, all knowledge is explicitly encoded in the algorithm by programmers using code statements, loops, and conditional statements. Therefore, knowledge must be known beforehand. In machine learning, knowledge is inferred from data. Knowledge is discovered.

Why do we need machine learning? Can't we just explicitly code algorithms for problems? There are two typical situations in which traditional programming cannot be used to solve problems. The first type is when it is not possible to encode all the rules needed to solve a problem. How would you encode rules for recognizing faces in photos? We don't even know

the rules we use in our minds to recognize faces so it would not be possible to encode rules. However, we can train computers to recognize key edges and regions of photos that are likely to be faces. The second type of situation in which traditional programming cannot be used to solve a problem is when the scale of the problem is too large. If a company has huge amounts of customer data it would take millions of human hours to find useful patterns in the data that could be then extracted programmatically. Machine learning algorithms can find patterns quickly in large amounts of data.

As we go through the material in this series of handbooks, you will learn several machine learning algorithms. These algorithms typically have statistical and probabilistic foundations which we will explore. However, beyond the theory and technique, machine learning is also a craft as well as a science. Throughout the series, we will point out some innovations, ideas, and techniques from this evolving craft.

Exercise 1.2 — Check your Understanding. There is a lot of new terminology in this chapter, let's take a breath and review.

- What is the difference between supervised and unsupervised learning?
- What is the difference between regression and classification?
- List at least 3 problems that machine learning could solve that traditional programming could not.

1.4 Terminology

Machine learning grew out of statistics and probability, computer science, as well as other fields. For this reason there are often multiple terms for the same thing. Let's start with names for data. The table below contains a sample data set (with headings).

GPA	Hours	SAT	Class
3.2	15	1450	Junior
3.8	21	1420	Sophomore
2.5	9	1367	Freshman

Table 1.1: Student GPA, Average Hours Studied/Week, SAT, Class

The table has 3 rows of data. Each row is a sample data point, also called an **example**, **instance**, or an **observation**. Each column in the table is an **attribute**, also called a **feature** or a **predictor**. The data has 4 features: GPA, average number of hours studied per week, SAT score, and classification. The first 3 are **quantitative**, or numeric, features while Class is a **qualitative** feature because it can only take on one of a finite set of values. Qualitative features are also called **factors** or **categorical data**.

If we want to learn GPA as a function of the other 3 features, we say that GPA is our **target**, or **response**, variable while the other 3 are **features** or **predictors**. If we want to predict SAT then SAT would be our target and all other columns our predictors.

Exercise 1.3 — Check your Understanding. What we will later call *tidy data* is data similar in structure to a spreadsheet.

- List two names for rows in the data.
- List two names for columns in the data.
- In what circumstance do we call a column of the data a predictor? A target?

1.5 Summary

This chapter has given a broad overview of the field of machine learning. As you go through this series of books you will gain new skills with every chapter and put them into practice in your own personal projects. Learning by doing is key to gaining useful skills.

1.5.1 Practice to Consolidate Skills

Learning doesn't just happen. Going through this series of books in a systematic way will provide a good foundation of machine learning skills. Beyond the course or the book, you will need a plan for continued learning. Here are a few suggestions for becoming a life-long machine learning learner.

- Identify your Purpose. Where do you see your machine learning skills in 5 years? How do these skills fit into an overall career path?
- Keep track your learning. I know I said that before but it's important enough to repeat. A notebook will avoid the frustration of knowing that you've done something before, but not remembering the file name to look up how you did it.
- Give yourself a work review. As you work weekly on your skills, schedule a periodic work review. Are you pleased with your progress? What areas need improvement? How do you plan to improve?

1.5.2 Next-Level Learning

Problem 1.1 — Create a Skills Showcase. I have had so many students get jobs and internships based on the code and projects they have done in my classes. A great way to showcase what you can do is by having your own GitHub page. You can get started by reading the documentation: `https://docs.github.com/en/get-started`, or watching a tutorial on YouTube.

If you don't have a GitHub page already, this may seem daunting, but take heart. This is the first skill you are learning in this series of notebooks, and it is a skill that will serve you well. A GitHub *repository* or *repo* for short, is just a collection of code. I have organized my repos around my books, but you may choose to organize them around topics or projects.

There are a couple of ways to manage your repositories. One is using a command-line interface with the **git** commands. An easier way in my opinion is to download GitHub for Desktop.

2. Learning Standard R

The goal of this first volume in the series is to learn how to use R for data analysis and machine learning. You might be wondering: why R? The main reason is that the early algorithms explored in the series, in this volume, have been used by statisticians for decades in R, and consequently the output of functions are more explanatory than the corresponding functions in Python libraries. R is simple enough for programming novices to learn and a breeze for computer scientists. Everything in R is built in. You can switch seamlessly from data cleaning, to data analysis, to creating plots, to running machine learning algorithms, to performing statistical analysis on your results.

R is open source, supported by a core group of contributors, with substantial online help from many sites. R gives us the ability to get work done quickly with minimal learning curves. R is used heavily in academia among data scientists, statisticians, and machine learning specialists. R is also commonly used in industry by companies with a lot of data like Google, Microsoft, Amazon, as well as traditional industries. R also gives you tools for easily sharing and communicating your results with html or pdf reports as well as interactive web sites.

In the next few sections you will learn:

- Where to find R and RStudio installation links and instructions
- R data structures and how to manipulate them in R syntax
- R data exploration functions and techniques
- R data visualization functions and techniques
- How to install and use R packages
- How to use R notebooks which combine text and code
- R control structures for more complex code
- Recommendations for R style

2.1 Installing R, RStudio

You will need to first install R and then install RStudio, a beautiful and powerful IDE for R. Both are available for Windows, Mac and linux.

Link to download and install R: `https://www.r-project.org/`

Link to download and install RStudio: `https://www.rstudio.com/`

Choose RStudio's free community edition. While you are on the site take a look around at some demos and tutorials. RStudio supports both R and Python, but we will be focusing on R in this volume. You will notice that RStudio is part of *posit*, which we will discuss later.

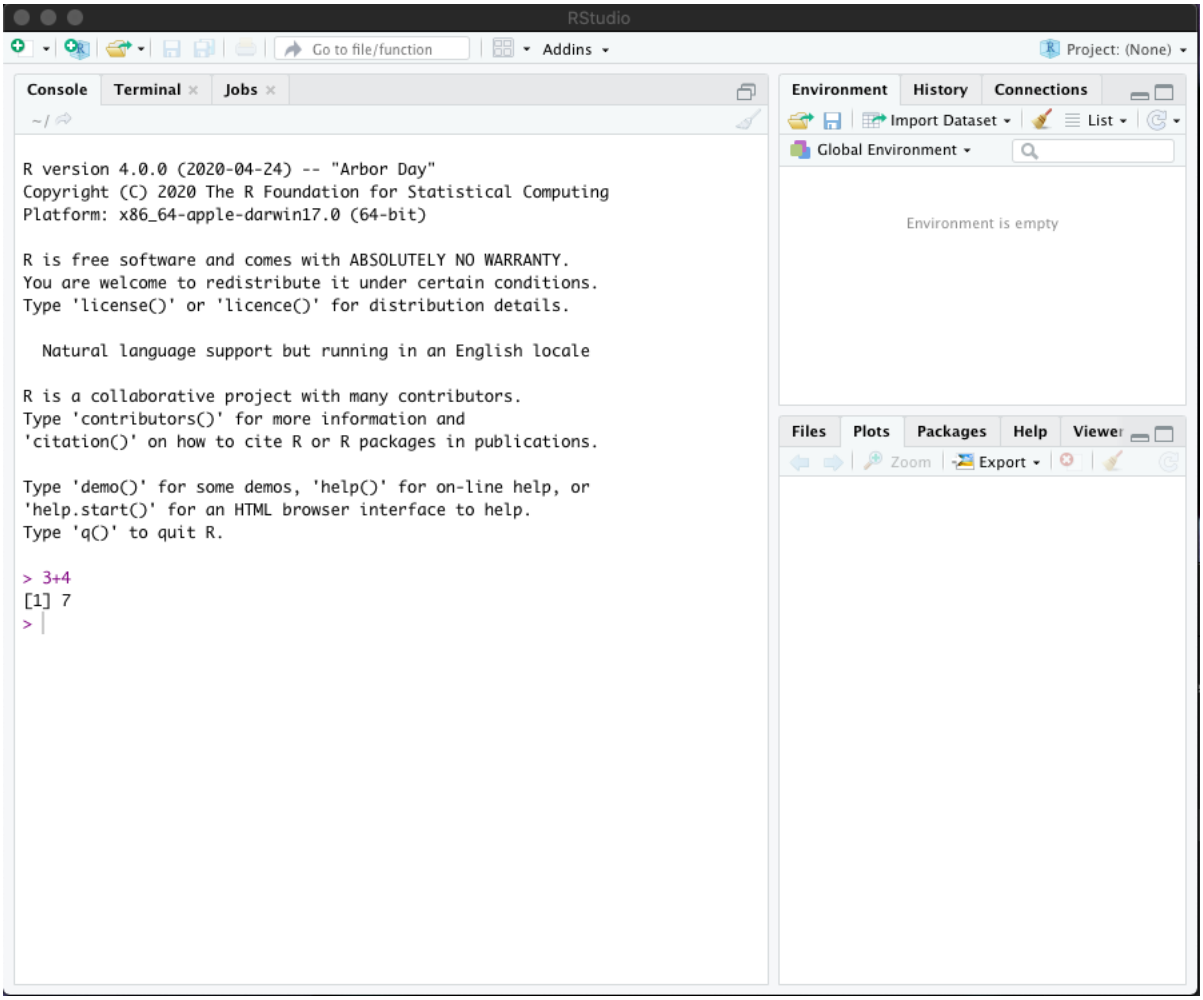


Figure 2.1: RStudio IDE

2.1.1 Getting Started in R

If you open up RStudio you will see a couple of panes on the right, and one large pane on the left, similar to Figure 2.1. For now, the pane on the left is the console. At the top of the console screen you will see some information about your version of R, then the interactive

prompt > waiting for you to do something. In the figure, we've typed in a simple expression: 3+4.

R is an interpreted language which makes it easy to experiment with ideas at the console. You can also save your code in notebooks or scripts, to run again and again. Although R is an interpreted language, most of the code is in C/C++ so it is fast. For now we will get to know R at the console. You may prefer to watch the video tutorial which covers the same material as this chapter. Whether you are reading or watching or both, please make sure you get practice typing in commands and seeing the results. You will learn more this way.

Type along with these examples in the console window as we go. Below, we typed an arithmetic expression at the prompt, hit enter and we see the results on the next line. There should be no surprises here, it's the usual operators with the usual order of operations.

```
> 3 + 4 - 1 * 8 / 2
[1] 3
```

But what is that [1] doing there in front of the output 3? That's just telling you that your result is a one-dimensional object. It's actually a vector of length 1. In R, pretty much everything is an object. Let's type in the same expression as above but this time save it to variable x. By the way, the up arrow displays the previous command, just like in a unix console. The less-than-dash, $<-$, is the assignment operator in R. We did not have to declare variable x. R figured out what type it is by what was stored in it. R has dynamic typing; if we change what type of object is in x later, R will not complain.

```
> x <- 3 + 4 - 1 * 8 / 2
> x
[1] 3
```

Now when we type in x at the console, R echoes back its contents. What is x? Let's ask R about it. There are many functions we can use to inquire about an object:

```
> is.numeric(x)
[1] TRUE
> class(x)
[1] "numeric"
> typeof(x)
[1] "double"
```

Note that numbers are stored as doubles by default. If we specifically want to store an integer we use the L indicator:

```
> x <- c(1L, 2L, 3L)
> typeof(x)
[1] "integer"
```

The c() function combines elements into a vector. A vector is a sequence of objects of the same type. The c() function will coerce its arguments to a common type, the most inclusive type in the vector:

```
> x <- c(1L, 1.2, "hi")
> typeof(x)
[1] "character"
```

2.2 R Data

We have seen in the examples above that data in R can be logical, numeric (integer or floating point), or character. Data is organized into objects of varying types. In this section we learn more about R objects that hold and organize data. The following table organizes the R data structures by their dimensions and whether or not all elements have to be of the same type.

Dimension	Homogeneous	Heterogeneous
1d	atomic vector	list
2d	matrix	data frame
nd	array	

Table 2.1: R Data Structures

2.2.1 Vector

A vector is a sequential structure with one or more elements of the same type. There isn't really a scalar in R, it's just a vector of length 1. Follow along with this code:

```
> x <- 1:10
> x*5
 [1]  5 10 15 20 25 30 35 40 45 50
>length(x)
[1] 10
>sum(x)  # try mean(), median(), range(), max(), min()
[1] 55
```

In the first line, x is assigned to be a vector from 1 to 10 with the sequence operator ":". In the second line we multiply each element by 5. In most programming languages you would need a loop to do this but note the simple syntax that allows us to do this in one line. There are many built-in functions in R. The next line shows length(). The last line returns the sum of each element of x. You may be wondering why it is 55 since we multiplied each element by 5 above. The reason is that we did not assign it back to x as in `x <- x*5`.

The vector x is numeric but we can have other types of vectors such as character or logical. When we type "x > 5" at the console, a logical vector returns with TRUE or FALSE for each element.

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

If we `sum(x>5)`, the TRUE values count as 1 and the FALSE values as 0 so we get a count of how many elements are greater than 5. Notice in the last line below that we can select those instances greater than 5 and replace only those with 0. Such power with such simple syntax!

```
> sum(x>5)
[1] 5
> x[x>5] <- 0
> x
 [1] 1 2 3 4 5 0 0 0 0 0
```

Subsetting vectors requires first learning one strange thing about R. It starts indexing at 1, not 0 like other programming languages. If you type `x[0]` at the console you won't get an error but some information about the vector, so it looks like that space is put to good use. Here are some indexing examples (anything after # is a comment):

```
> x <- 1:10
> x[0]
integer(0)
> x[2:4]      # use : for a range of elements
[1] 2 3 4
> x[c(2:4,8)] # use c() for noncontiguous elements
[1] 2 3 4 8
> x[11]
[1] NA
```

Notice above that we did not get an error when we had an index out of bounds, it just gave us an NA, for "not available".

To check the type of a vector you can use:

- typeof(x) – returns type
- is.integer(x) – returns boolean
- is.double(x) – returns boolean
- is.numeric(x) – returns TRUE for integer or double
- is.character(x) – returns boolean
- is.logical(x) – returns boolean

To convert the elements of a vector from one type to another you can use functions such as `as.integer`, `as.double`, etc.

Exercise 2.1 — Vectors. The term vector can be a bit puzzling at first. Clear your mind of its meaning in physics. In R it is merely a sequential container for variables that can be indexed.

- Create a vector i of values 5 through 12.
- Experiment with vector arithmetic. Add a scalar to each element of your vector. Multiply each element by a different scalar.
- Find the mean and median of your vector.
- Determine the type of data in your vector, and change it to a different type.

■

2.2.2 Lists

A list is an ordered collection of objects not necessarily of the same type. Lists can contain other lists as elements. Lists are often used as holders for things returned from functions. List elements are selected with double square brackets:

```
z <- list(1,2,3)  # create a list
z[[1]]
[1] 1
y <- list('a', TRUE, z)
> typeof(y)
[1] "list"
> length(y)       # y is a list of length 3
[1] 3
> length(y[[3]])  # the 3rd element is also a list of length 3
[1] 3
```

Lists v. Vectors. A list is a generic vector where elements do not have to be of the same type. For this reason, to check if an object is a vector, don't use `is.vector()` but the following:

```
x <- 1:5      # create a vector
is.atomic(x)  # check if x is a vector
[1] TRUE
```

You can check if a list has lists as elements with the `is.recursive()` function. The `unlist()` function converts a list into a 1-dimensional atomic vector, coercing all elements into the most general type.

```
> w <- unlist(y)
> w
[1] "a"    "TRUE" "1"    "2"    "3"
> typeof(w)
[1] "character"
```

2.2.3 Matrix

A matrix is a 2-dimensional object with elements of the same type. The code below creates a sequence from 1:10 and stores it in a matrix of 2 rows.

```
> m <- matrix(1:10, nrow=2)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Notice the handy reminders showing how to index at the top of the columns and the left of the rows. Practice indexing matrix m. Indices are `[row, col]`.

We will mainly use vectors and data frames (discussed below) in machine learning but we will use matrices from time to time.

2.2.4 Arrays

Arrays are similar to matrices but can have more than 2 dimensions. Like matrices, all elements must be of the same type.

2.2.5 Data Frames

A data frame is a 2-dimensional structure where each column can be of a different type. Most of the time we will load a data frame from disk but we can create one manually by creating 3 vectors that we combine with cbind (column bind):

```
> x <- c(1,2,3)
> y <- c(1.1, 2.2, 3.3)
> z <- c('a','b','c')
> df <- data.frame(cbind(x,y,z))
> df
  x   y z
1 1 1.1 a
2 2 2.2 b
3 3 3.3 c
```

In a data frame, each column is a vector of a specific type. All the columns are of the same length. Observe in the code below that we can access these column vectors with the dollar operator and that we can also change the column names.

```
> df$x
[1] 1 2 3
Levels: 1 2 3
> colnames(df) <- c('Ticket', 'Discount', "Section")
> df
```

```
  Ticket Discount Section
1      1      1.1       a
2      2      2.2       b
3      3      3.3       c
```

We can read in a text file or csv file from disk as shown below. If you want to know more about the options for read.csv use the command at the bottom of the following code example. The str() functions tells you about the structure of the data frame, its columns, and what type of data it contains, as well as how many observations and variables.

```
> df <- read.csv("titanic.csv")
> str(df)
'data.frame': 1309 obs. of  14 variables:
 $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : chr  "Allen, Miss. Elisabeth Walton"  ...
 $ sex      : chr  "female" "male" "female" "male" ...
 $ age      : num  29 0.917 2 30 25 ...
  . . . and more . . .
```

There are also many built-in data sets in R. You can find out about them by typing data() at the console.

2.3 Data Exploration

You can load a built-in data set with the data() function and then look at a few rows with head() or tail(). Comments start with # so you don't need to type those in when you are working at the console.

```
> data(airquality)
> head(airquality, n=2) # see the first two rows
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
```

Here are several data exploration functions you should explore at the console, replacing "df" with the name of your data frame, such as "airquality":

- names(df) lists the column names
- dim(df) gives the row, col dimensions
- summary(df) gives summary statistics for each column
- str(df) gives row and column counts, information for each column
- head(df) and tail(df) give the first and last 6 rows by default

Missing items in a data frame are typically encoded as NA. This can cause some problems for built-in functions as shown below. Notice the mean() function returned NA until the na.rm=TRUE parameter was added.

```
> sum(is.na(airquality$Ozone))  # count the NAs
[1] 37
> mean(airquality$Ozone)
[1] NA
> mean(airquality$Ozone, na.rm=TRUE)
[1] 42.12931
```

A trickier problem is deciding what to do with NAs before we run data sets through machine learning algorithms. Some algorithms may not work and others may give less than optimal results with NAs. One option is to delete rows that have NAs but this could be a problem if your data set is already small. Another option is to replace NAs with a mean or median value as shown next. First we copy the data set to another variable so we won't alter the original.

```
> df <- airquality[]
> df$Ozone[is.na(df$Ozone)] <- mean(df$Ozone, na.rm=TRUE)
> mean(df$Ozone)
[1] 42.12931
```

R syntax can be quite compact and nested; it takes a little getting used to. The syntax `df$Ozone[is.na(df$Ozone)]` above selects only those rows in df in which Ozone is NA. Only these will be replaced by the mean of the column.

Exercise 2.2 — Data Exploration. Type `data()` at the console to pop up a list of build-in data sets.

- Select a built-in data set that interests you.
- Examine the data set with at least 3 built-in functions such as summary(), str(), head().
- Read the documentation for the data in the help pane (type '?x' at the console where x is the name of the data set).
- Write a brief description of the size of the data, the columns and their data types, what the columns indicate.

2.4 Visual Data Exploration

In addition to exploring data with the R commands described in the last section, and exploring the data with built-in statistical functions, we can also explore data with graphs. Type `demo(graphics)` at the console to see the variety of graphs you can create in R. The next chapter gives a concise overview of basic graph functionality in R. Next we continue looking at airquality with R graphical functions.

The plot() function is most often used to plot points as in `plot(x, y)`, where the first argument is plotted on the x axis and the second on the y axis. If only one argument is given, as in `plot(y)`, the x axis will be a numbering vector. Type the following at the console to see the graphs:

```
hist(airquality$Temp)
plot(airquality$Temp)
plot(airquality$Temp, airquality$Ozone)
```

Here are some useful arguments for modifying graphs:

- pch - specifies the symbol to use for points; 1 is the default
- cex - specifies symbol size, 1 is the default, 1.5 is larger, 0.5 is smaller
- lty - specifies line type, default is 1
- lwd - specifies line width, default is 1
- col - color of graph element; colors can be specified by name, number, hex; col=2 and col="red" are the same
- xlab - label for x axis
- ylab - label for y axis
- main - main label

We will learn more about graphs in R as we go. The following link provides a good overview of R graph parameters: `https://www.statmethods.net/advgraphs/parameters.html`

Below is an example of using some of these parameters in code. Figure 2.2 shows the resulting plot. The plot() code says to plot Ozone on the x axis and Temp on the y axis. These plot points are shown with pch=16 (see the statmethods link above for other options), color blue, and 1.5 times the regular size. We also specified a main label as well as x and y axis labels.

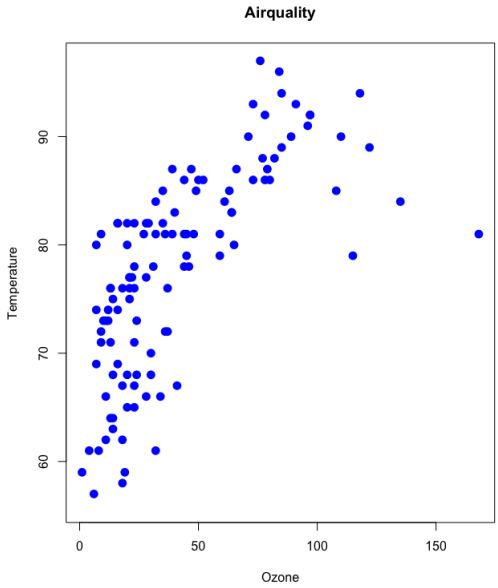


Figure 2.2: Airquality

```
plot(airquality$Ozone, airquality$Temp, pch=16, col="blue", cex=1.5,
    main="Airquality", xlab="Ozone", ylab="Temperature")
```

Often we want to know if columns are correlated with other columns. A high positive correlation between x and y would look like a line of slope 1. A high negative correlation would look like a line of slope -1. Correlation is quantified on a scale of -1 to $+1$, with values closer to the 1s indicating strong positive or negative correlation and values closer to 0 indicating weak or no correlation. The code below shows how to generate a correlation matrix with the cor() function, and how to create a graph with the pairs() function. See if you can identify correlations in the graphs.

```
> cor(airquality[1:4], use="complete")
             Ozone    Solar.R       Wind       Temp
Ozone    1.0000000  0.3483417 -0.6124966  0.6985414
Solar.R  0.3483417  1.0000000 -0.1271835  0.2940876
Wind    -0.6124966 -0.1271835  1.0000000 -0.4971897
Temp     0.6985414  0.2940876 -0.4971897  1.0000000
> pairs(airquality[1:4])
```

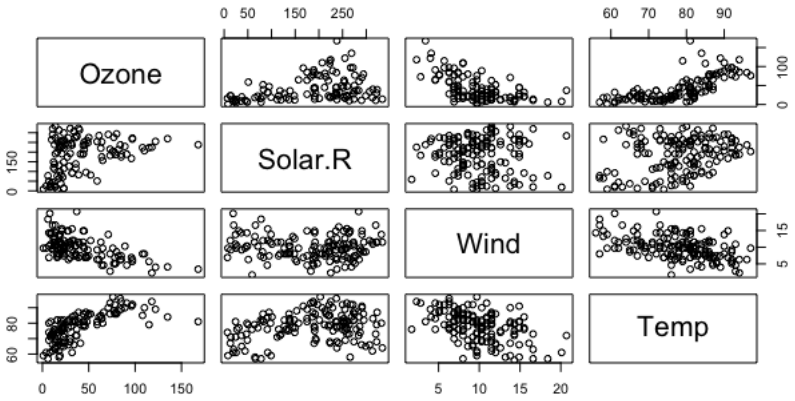


Figure 2.3: Correlation Pairs for Airquality

The cor() code above ignores NAs because of the use= parameter. Remove that extra parameter and compare your results to the output above. The pairs() command requested to look at only columns 1:4. The highest positive correlation we see in the cor() output is between Temp and Ozone. The graph confirms this correlation: the higher the temp, the higher the Ozone. The largest negative correlation is between Temp and Wind. Looking at its plot we see a definite downward trend. Solar.R and Wind have a correlation much closer to 0 and we see in the plot that there really is not much of a discernible pattern there.

Exercise 2.3 — Data Visual Exploration. For this exercise, use the same data set you selected in the previous exercise.

- Plot at least 3 of the columns. What did you learn about the data?
- Plot one variable on the x axis and one on the y axis. What did you learn about the data?
- Create a histogram of one of the columns. What did you learn about the data?

■

2.5 Factors

We have seen vectors of various types: integer, numeric, character, and logical. There is one more type of vector that will be important to us. Factors are used to encode qualitative data. A factor vector is a vector of integers with an associated vector of labels. Internally it is stored as integers but for our benefit they display as labels. Look at the Titanic data.

```
> df <- read.csv("data/titanic.csv", na.strings="NA", header=TRUE)
> str(df)
'data.frame': 1309 obs. of  14 variables:
 $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : chr  "Allen, Miss. Elisabeth Walton"  ...
 $ sex      : chr  "female" "male" "female" "male" ...
 $ age      : num  29 0.917 2 30 25 ...
. . .
```

The above code assumes that you have the data/csv file in the same directory you are working in. We used the parameter `na.strings="NA"` to tell R to fill missing cells with NA. This data is a bit messy because some things that should be factors, like pclass or survived, are not. Since R 4.0, strings are not read as factors. If you want strings read as factors use the `stringsAsFactors=TRUE` parameter. We can convert to another data type as shown below. Notice we can tell R about the factor levels we want if we use factor() instead of as.factor().

```
df$pclass <- as.factor(df$pclass)
df$sex <- factor(df$sex, levels=c("male", "female"))
```

If you rerun the str() function on the data you will see that pclass and sex are now factors. We can see how many levels a factor has with the levels() function, and see the encoding with the contrasts() function as shown below.

```
contrasts(df$sex)
       female
male        0
female      1
```

```
> contrasts(df$pclass)
  2 3
1 0 0
2 1 0
3 0 1
```

For sex we only need one variable to encode the two genders in the data. The contrasts for pclass shows that we need 2 variables to encode 3 classes. The base case will be class 1. R will create 2 *dummy variables* for classes 2 and 3. We will see the importance of these when we get to machine learning.

2.5.1 Adding a Factor Column to a Data Frame

The following code adds a new column to our data frame for airquality. First it makes all items = FALSE, then makes those with a temperature over 80 to be TRUE. Finally we coerce it to be a factor. The first line copies the data frame.

```
> df <- airquality[]  # copy the data set
> df$Hot <- FALSE
> df$Hot[df$Temp>89] <- TRUE
> df$Hot <- factor(df$Hot)
> df$Hot[40:46]
[1] TRUE  FALSE TRUE  TRUE  FALSE FALSE FALSE
Levels: FALSE TRUE
>plot(df$Hot)
```

Here are some plots of the Hot column, arranged in a 1x3 grid with the par() function. Plotting the column by itself just gives us a visual of the distribution of Hot and not Hot. The cdplot() gives a conditional density of Hot (light grey) and not (black) across the x axis which represents temperature. The third plot is a box plot in which the heavy middle line in the box denotes the median, the box itself indicates the IQR and the horizontal lines at either end of the dashed line indicate min and max, excluding suspected outliers which are dots beyond that line.

```
> par(mfrow=c(1,3))
> plot(df$Hot)
> cdplot(df$Temp, df$Hot)
> plot(df$Hot, df$Temp)
```

The graphs shown in this chapter are rather plain. We can add headings, labels, color and more. Read more about graphical parameters here: `https://www.statmethods.net/advgraphs/parameters.html`

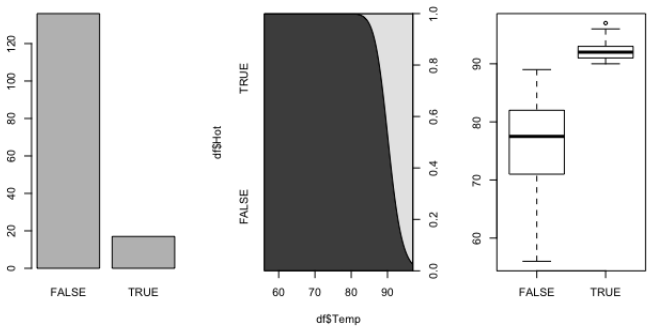


Figure 2.4: Airquality

Exercise 2.4 — Factors. For this exercise, use the same data set you selected in the previous two exercises.

- Were any of the columns factors? What values could the column have?
- Create a new column in the data set, converting a quantitative column to a qualitative column by using a cut-off point such as mean or median, then converting it to a factor.
- Create at least two graphs using the new column.

2.6 R Notebooks

Here are a few things to keep in mind about R:

- R is case sensitive
- The period has no special meaning unlike many other languages
- The dollar sign acts somewhat like the period in other languages
- The hashtag starts comments
- White space is ignored
- The R workspace stores all the objects you create in a session. You can save it, but for now you should always choose No when it asks you to save the workspace upon exiting RStudio.

If you end up with items in your workspace that you don't want, you can list them with ls() and remove things with rm(). The second command below shows how to remove everything from the workspace environment.

```
> ls()
[1] "airquality"
> rm(list=ls())
```

R is a powerful language in its own right. However it becomes truly awesome when we load packages built for specific purposes. These packages are available on CRAN, the Comprehensive R Archival Network. We install packages at the console with the install.packages()

command. You only have to install once. We load packages into our working environment only once per script with either the library() or require() functions.

Before we get started with an R notebook, we should note that you can also create a simple R script. These are text files that end in .R and can be run from the console or from within RStudio. Go to File then New File, then R Script. You will see a blank document in RStudio on the left. You simply type in code, perhaps try out the code below. Then you can run the script from the Run button at the top. The output of the head() function will display below the script. The plot will display in the lower right. Give that a try just so you are comfortable with writing an R script.

```
# simple script to demo R code

# take a look at the built-in cars data set
head(cars)

# plot stopping distance Y axis, speed X axis
plot(cars$speed, cars$dist)
```

RStudio (aka posit) has released a newer upgrade to Rmarkdown notebooks called Quarto. This volume will stay with Rmarkdown instead of Quarto because the overhead is simpler and most importantly, the html notebooks display better in GitHub. Should you wish to convert your Rmarkdown notebooks to Quarto documents, the process is very simple, see the documentation: `https://quarto.org/docs/faq/rmarkdown.html`

We are going to focus on R notebooks which intersperse text commentary and code. Comments in an R script as shown above start with a hashtag. Comments in a notebook will be much more extensive. In RStudio, go to File, New File, then R Notebook. Save your file. I recommend creating a folder to keep all your notebooks in, then always loading from that folder by double-clicking on it.

Once you create the R notebook you will see some YAML code at the top. You can also add an author line, and you should change the title. YAML is a recursive acronym for YAML Ain't Markup Language. This YAML tells RStudio to what kind of output to create, in this case it will make an html notebook that you can upload to your website or github.

```
title: "R Notebook"
output: html_notebook
```

You'll notice white portions for text and grey portions for code in the boiler plate notebook that is created for you. Read through the standard text which provides some useful tips on things like how to add new code chunks and get started with markdown text. You can type regular text in the white portions but markdown is worth learning because you will encounter it in many situations beyond R. There is a nice cheat sheet for markdown in RStudio's web site. The really nice thing about markdown is that you can format as you type without taking your hands off the keyboard. In the figure below, the three hashtags denote a level-3 heading.

When you create a pdf or html file from this notebook, you won't see the hashtags, just the text rendered as a level-3 heading.

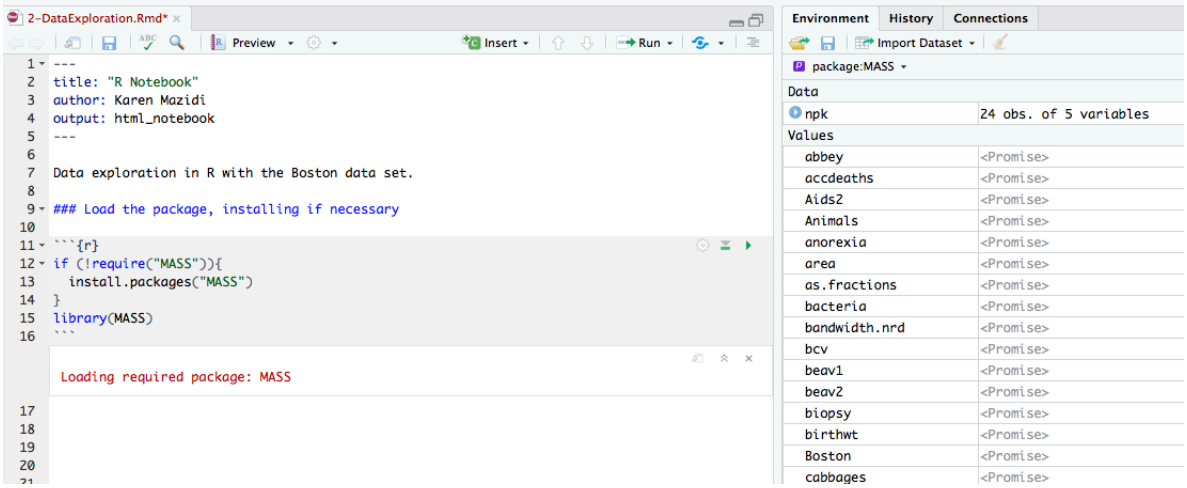


Figure 2.5: RStudio Environment

Figure 2.5 shows the RStudio environment with the white portions for text markdown and the grey portions for code. The full notebook is available on the github site for this book. You should practice making your own notebook similar to the sample. Let's point out a couple of things before we move on. First, once the MASS package was loaded, you can see it in the environment in the upper right. This environment is a great way to see your variables as you go. It is a great help in debugging. Second, notice the Preview button above the notebook. You can use this to "knit" or render your file into html or pdf or other formats. This option is also available under the File menu.

2.7 Control structures

We have seen earlier that a lot of things we might use a loop for in other languages can be done in simple R syntax. However there will be times when we want to code loops, conditionals, and functions. We describe those in this section, in an example getting you started using an R notebook.

This section uses more advanced R code so it is fine to skim through this section and come back to it when you need to use loops, conditionals, or functions. This code will become second nature only with practice. The notebook is in the github site so you can refer to it as you need it.

If you wish to go through this section now, open RStudio and go to File->New File->R Notebook. Get rid of the text in the white areas. Change the title at the top of the file. In the grey code box, remove the code that is there and type the code below. This is our first look at an **if** statement. R is very punctuation-heavy. Indents are for style and readability only. Notice that the condition is surrounded by parenthesis, and the body of the **if** statement is surrounded by curly braces. The opening curly brace is on the same line as the condition, and the closing

curly brace is on its own line. This is a standard style in R, but we will talk about style more later.

```
if (!require("mlbench"){
    install.packages("mlbench")
}
library(mlbench)
data(PimaIndiansDiabetes2)
str(PimaIndiansDiabetes2)
```

The **if** statement will check if the *mlbench* library is already installed. If not, it will install it. The last three lines load the library, then the data set, then look at the data set. Notice that installing the library does not load it into memory.

You can run this chunk of code with the green arrow on the right side of the box. The package mlbench contains several benchmark ML data sets. This code first checks if package "mlbench" is installed. If it is already installed, the package will be loaded into memory. If the package is not installed, it will execute the install.packages() function to install it. Installing a package should just take a minute. If R asks you for a mirror site in a pop-up, just pick one. Once the package is installed, from then on you just load the package into memory in future notebooks with library() or require().

In the upper right pane of RStudio, click on Environment then Global Environment. You should see that the data has been loaded into memory. You should see that it has 768 observations and 9 variables. The str() command you ran above will output information about each variable. You can learn more about this data set by typing `?PimaIndiansDiabetes2` at the console.

From the str() function above, you can see that there are a lot of missing values denoted by NA. Create a new code chunk at the botton of your notebook by clicking on Insert->R at the top of the notebook window. Type in the following code and see the results. The sapply function applies a function to data. Here we have an anonymous function coded on the fly to sum NAs. The sapply() function will apply this to each column. We have a few NAs for glucose, mass, and pressure, but a lot for triceps and insulin. If we just omit all rows with NAs that will cut our data in half. An alternative to just deleting them is to fill them with either the mean or the median of the column. In the code section 2.7.1 shown below, we use an if-else to either calculate the mean or median of a vector. Then we write a function to fill NAs of a vector with either the mean or the median. Whether or not it is a good idea to fill missing values in this way may depend on how you are using the data. Always document any changes you made to your data.

```
> sapply(PimaIndiansDiabetes2, function(x) sum(is.na(x)))
pregnant  glucose pressure  triceps  insulin     ...
       0        5       35      227      374        ...
```

Figure 2.6: Second Code Chunk and Results

2.7.1 if-else

Within the function is an if statement. The if statement in R has this form:

```
if (condition) {statements if true} else {statements if false}
```

The ( ) around the condition is required, as are curly braces around the statements. Statements should occur on individual lines for readability, although R will allow multiple statements separated by a semicolon. We will discuss R style preferences below, but note that the opening { is at the end of a line and the closing } is on a line by itself. The if-else statement below lets us use either mean or median, depending on the choice sent to the function.

We will see examples of the ifelse shortcut throughout the book. Here is the format:

```
ifelse(cond, true, false)
```

Exercise 2.5 — If Statements. In R we can have if statements, nested if statements, and the ifelse shortcut.

- Load the PimaIndiansDiabetes2 data set in memory.
- Write and test an R statement to return a TRUE FALSE vector for glucose values over the mean.
- Write and test an if-else statement to print "average age over 30" or "average age less than 30" based on the mean of the age column.
- Write and test an ifelse statement to return a TRUE FALSE vector if age is greater than 30

■

2.7.2 Defining and Calling Functions

The code below shows a user-defined function. The definition format is:

```
name <- function(args) {statements}
```

Code 2.7.1 — Function Example. Third Code Chunk

```
fill_NA <- function(mean_med, v){
  if (mean_med == 1){
    m <- mean(v, na.rm=TRUE)
  } else {
    m <- median(v, na.rm=TRUE)
  }
  v[is.na(v)] <- m
  v
}
```

The name of the function above is `fill_NA` and it expects two arguments. The first argument is a flag indicating whether the function could compute mean or median. The second argument is the input vector v.

Statements to call a function simply have the name of the function and the arguments. The result of the function replaces the original values of those vectors. The last line of the third code chunk handles the remaining missing value which are few. The data set df will have 724 observations.

The next-to-last line of our function replaced all NAs in the vector with the mean (or median). Notice that there is no "return" statement. The result of the last statement in the function is what is returned, which in this case is our updated vector.

```
# make a new data set with NA's filled
df <- PimaIndiansDiabetes2
df$triceps <- fill_NA(1, df$triceps)
df$insulin <- fill_NA(1, df$insulin)
df <- df[complete.cases(df),]  # omit rows with NAs
```

For an additional example of a function in R, let's look at a recursive function for the familiar Finonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...

Code 2.7.2 — A Recursive Function. Fibonacci Sequence

```
fib <- function(n){
  if (n <= 1){
    return(n)
  } else {
    return (fib(n-1) + fib(n-2))
  }
}

# print a sequence
for (i in 1:8)
    print(fib(i))
```

Try this out on your computer. Note in the above code that we used return(). However, the function would work correctly without the surrounding return(). For example, instead of `return(n)` just `n`.

2.7.3 Loops

R has for-loops and while-loops. The formats are:

```
for (condition) {statements}
while (condition) {statements}
```

Figure 2.7 shows an example for-loop. The loop creates 3 linear models with function lm(). We will learn more about this in the chapter on linear regression. The 3 iterations of the for loop will use formula "glucose~df[,cols[i]]" to build 3 models: glucose as a function of mass, glucose as a function of age, and glucose as a function of the number of pregnancies, because these are the columns selected in variable cols. These three models are stored in a list.

First we plot mass on the x axis and glucose on the y axis. Then we plot 3 ablines, one for each stored model. The abline function can be used to plot the regression line as we are doing here, but can also be used to plot straight lines. We make each line a different color by just letting color =i, our index. Finally we add a legend using the same color codes as the ablines.

```
```{r}
cols <- c(6,8,1)
plot(df$mass, df$glucose, main="PimaIndianDiabetes2")
for (i in 1:3){
    model <- lm(glucose~df[,cols[i]], data=df)[1]
    abline(model, col=i)
}
legend("topright", title="Predictors", c("Mass", "Age", "Pregnant"), fill=c(1,2,3))
```
```

Figure 2.7: Fourth Code Chunk

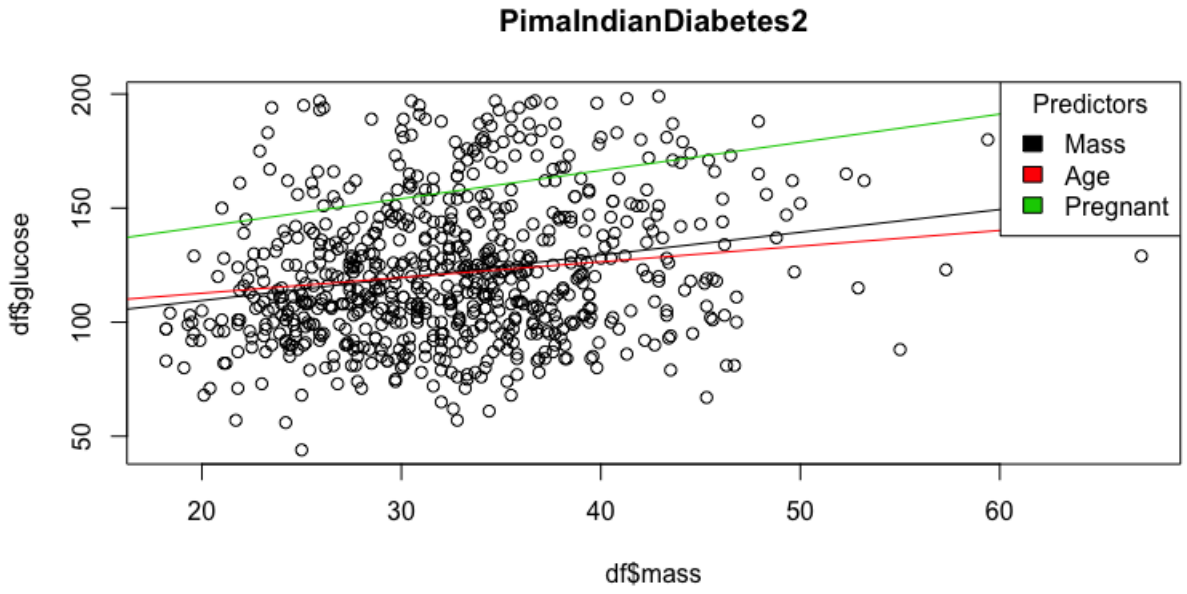


Figure 2.8: Graph with Regression Lines

Next, we demonstrate the ifelse() shortcut. The ifelse() in Figure 2.9 checks if insulin is greater than 155. If this is true, the element will be 1 otherwise it will be 0. The

ifelse(df$insulin>155,1,0) creates an integer vector of 1s and 0s. This is then converted to a factor vector by surrounding it with the factor() function. This nesting of function is common in base R code. It is a little cumbersome to unpack. Later in this part of the book we will look at a pipe syntax that is more readable. You will see code out in the wild written in both styles, so it is good to get used to reading them.

In the first line of the 5th code chunk we are adding a new column to the data frame. In the second line, we plot the observations again, this time color coding those with high insulin as red, and those that do not have high insulin as blue. The bg= argument indicates what fill color is used for the points. We have a vector of "blue" and "red". How these colors are selected is controlled by the unclass() function which converts the "large" column factors to 1 or 2. So observations with large insulin values display as red and those with low values display as blue.

```
```{r}
df$large <- factor(ifelse(df$insulin>155,1,0))
plot(df$mass, df$glucose, pch=21, bg=c("blue","red")[unclass(df$large)])
```
```

Figure 2.9: Fifth Code Chunk

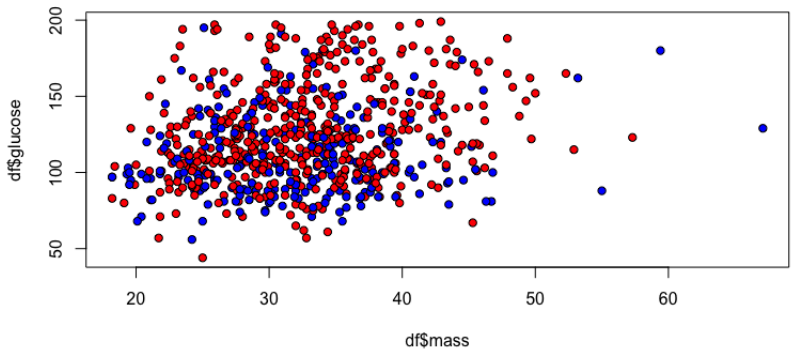


Figure 2.10: Graph with Color Coding

Exercise 2.6 — Control Statements. Looping statements and functions.

- Load the PimaIndiansDiabetes2 data set in memory.
- Write a while statement to print the first 10 ages in the data set.
- Set variable age_sum to 0. Write a for statement to add up all the ages. Then write a single line of R code to do the same thing.
- Write a function, that given a vector, prints the mean and median.

2.8 R Style

Google has its own R style guide available here:

`https://google.github.io/styleguide/Rguide.xml`

Another style guide, written by Hadley Wickham, is available here: `http://adv-r.had.co.nz/Style.html`. Hadley Wickham is the Chief Scientist at RStudio, and an influential R developer, having created several amazing open-source packages such as ggplot2, dplyr and more. The point is not to follow one style guide rather than the other, but to be consistent in your code. It makes it easier to read, even for you.

Here are a few recommendations from Hadley Wickham's style guide:

- variable and function names should be lowercase, using underscore to separate names
- variable names generally should be nouns and function names should be verbs
- try to use names that are concise but meaningful
- don't use names of existing functions; R will let you override them
- put a space around operators, except :
- otherwise don't add unnecessary spaces
- opening curly braces should not be on their own lines
- closing curly braces should go on a new line unless followed by else
- indent with 2 spaces, exception: function arguments

2.9 Summary

This chapter provided a fast-paced introduction to R. Don't expect to retain everything in the chapter right now, you can refer back to it as you need it. You will learn more R as we go, but now you already know enough to get started with machine learning.

In the github are full notebooks of the data exploration and R control structure examples in this chapter. Make sure you understand the R code in these notebooks. Link: `https://github.com/kjmazidi/Machine_Learning`. I have also created short video tutorials on my youtube channel which will provide a good review of what you should know from this chapter: `https://www.youtube.com/c/JaniceMazidi`

The RStudio website has a page full of useful cheat sheets: `https://rstudio.com/resources/cheatsheets/`. The Base R cheat sheet summarizes many aspects of R that have been covered in this chapter. The following Quick Reference supplements the cheat sheet with some additional code covered in this chapter.

This chapter showed several built-in functions for data exploration of a data frame:

- str(df) - gives an overview of the structure of the data; don't think of it as 'string'
- dim(df) - gives the dimensions of the data frame
- summary(df) - gives summary statistics for each column; summary(df$col) gives summary statistics for just column 'col'
- head(df) and tail(df) - look at the first or last few rows
- names(df) - list column names
- cor(df) - print a table of correlations
- pairs(df) - display a table of graphical representations of correlations

Other functions are typically used to provide information about a column: mean(), median(), sum(), max(), min(), and range().

2.9.1 Quick Reference

Dealing with NAs

Reference 2.9.1 Count the number of NAs

```
sum(is.na(df$col))
```

Reference 2.9.2 Count NAs by column

```
sapply(df, function(x) sum(is.na(x)))
```

Reference 2.9.3 Run stat functions ignoring NAs

```
mean(df$col, na.rm=TRUE)
# works with sum(), min(), max(), and more
```

Reference 2.9.4 Replace NAs with the mean

```
df$col[is.na(df$col)] <- mean(df$col, na.rm=TRUE)
```

2.9.2 Factors

Reference 2.9.5 Check the factor encoding

```
# using levels
v <- factor(c(1, 0, 1, 0))
levels(v)  # output levels
[1] "0" "1"
```

Reference 2.9.6 Change the factor encoding

```
# change factor encoding
levels(v) <- c("male", "female")
```

Reference 2.9.7 Using contrasts()

```
# contrasts gives integer and char encoding
contrasts(v)

            female
male        0
female      1
```

2.9.3 Practice to Consolidate Skills

If you took your time and coded along with either this chapter or the video on YouTube, you are well on your way to coding comfortably in R. You have this material to refer to, and there are also R and RStudio cheat sheets in the GitHub repo for this volume.

R is free to roam outside the confines of RStudio. With extensions, you can run R code in VSCode and many other IDEs. RStudio is ideal, however, because of all the information it gives you in one place: your code, your data, your plots, and more. Of all the IDEs I use, it remains my favorite for these reasons.

Problem 2.1 — Data Exploration in R. The only way to learn code is by coding, so let's dive in.

- Create a new notebook in RStudio, a good name might be Data Exploration 1. Fill in the YAML as you wish, and start the notebook off with headings of various levels stating the purpose of the notebook and what it demonstrates.
- Preface each code chunk with markdown description of what the code does.
- Load the MASS library and data frame Boston.
- Write a function that outputs sum(), mean(), median(), range() for a given input vector.
- Write a function that outputs the covariance and the correlation between two input vectors.
- Plot medv as a function of rm.
- Observe something unusual about the top of the graph. Explore the medv column to determine what is going on. Write some commentary about what you observe, and how this strange aspect of the data might be remedied.
- Output your notebook as an html file and include it in your GitHub repo.

Problem 2.2 — R Comparison with C++. R is an interpreted language and interpreted languages are generally slower than compiled languages. C++ is commonly used for custom machine learning coding due to its speed.

- Create a C++ program (or comparable language) to replicate the results of your R Notebook above (except the graphs). C++ is commonly used for custom machine learning coding due to its speed.
- How would you compare the *time* it took you to create the R notebook compared to the program you wrote from scratch?
- How would you compare the *look* of the html output from above compared to your program? And how sharable is the R code?

2.9.4 Next-Level Learning

You can use RStudio for Python code as well, making it even more worthwhile to learn. In 2022, the name posit was adopted, as they stated in a blog post:

> Posit is a real word that means to put forth an idea for discussion. Data scientists spend much of their day positing claims that they then evaluate with data. When considering a new name for the company we wanted something that reflects both

> the work our community engages in (testing hypotheses!) as well as the scientific aspiration to build ever- greater levels of knowledge and understanding.

Perhaps an understated reason on that blog post[1] was that people were using both R and Python and the platform needed to address that.

Explore the `https://posit.co/` website to learn more about how posit is positioning itself to be a significant player in data science, research, and technical communication. Later in this volume we will explore Shiny and Quarto, two exciting products for data dashboards and more.

[1] `https://posit.co/blog/rstudio-is-becoming-posit/`

3. Data Visualization in R

R has great data visualization functions that are quite simple to use. Modern R has extended the visualization capacities of R with the `ggplot2` package, discussed in the next chapter. In this chapter we give an overview of the types of graphs you can create using standard R. To get a feel for the R's graphic capabilities, type `demo(graphics)` at the console. This chapter will get you familiar with how data can be visualized in R and importantly, what types of graphs are most important for different data types. You might want to skim over this chapter to get the big picture, then refer back to it as you make your own graphs in the context of machine learning solutions later in the book.

In this chapter we first discuss data visualization techniques for a single column of a data frame. Then we discuss data visualization for two columns, two dimensional visualization. Throughout the chapter we use the Titanic data for the sample graphs. As usual, you can find the code for all the graphs and full-color figures on the github. Additionally, there are two good resources that you should bookmark:

- An overview of graphical parameters from the stat methods site [1]
- Color names for colors in R provided by Professor Tian Zheng at Columbia [2]

You can display graphs individually, or display them in grids. The par() function is used to set up a grid. The following code shows how to save the original parameter settings in a variable called opar, then plot graphs in a 1x2 grid, and finally restore the original parameter settings. The par() function can be used to set up any grid pattern you like, such as 3x2, etc. The graphs will be placed left-to-right, top-to-bottom, in the grid.

[1] `https://www.statmethods.net/advgraphs/parameters.html`
[2] `http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf`

```
opar <- par()      # copy original settings
par(mfrow=c(1,2))  # set up 1x2 grid
hist(...           # make a plot
plot(...           # make another plot
par(opar)          # restore parameter settings
```

3.1 Data Visualization of One Variable

In the online notebook, we first load the Titanic data and do a little clean-up. A given column in a data frame is just a vector. We first look at ways to plot quantitative vectors and then look at visualizing qualitative vectors.

3.1.1 Quantitative Vectors

The most common graph type for one quantitative variable is the histogram. You can specify the bins, but in the graph below, using the default settings worked out well. Type `?hist()` at the console to see all the parameters you can modify. Another plot type that is appropriate for quantitative data is a simple scatterplot. Since we only have one variable, R will supply row index numbers for the x axis. The code and graphs are shown below. Not that the plots are displayed side-by-side using par(), the original parameter setters were store at the top of the code block and restored at the end or the block.

Code 3.1.1 — Titanic data. Histogram and Scatterplot.

```
opar <- par()    # copy original settings
par(mfrow=c(1,2))

hist(df$age, col="slategray", main="Age of Titanic Passengers",
    xlab="Age")
plot(df$age, pch=21, cex=0.75,
   bg=c("snow", "slategray")[unclass(df$survived)], ylab="Age",
   main="Age (White Deceased)")

par(opar)
```

You can create the histogram with code as simple as `hist(df$age)` to get the visualization you need for the data. Later you can add the colors, titles, etc. For the scatter plot we added color, conditioned on the survival status of the passengers. The unclass() function converts the survived 0s and 1s into 1s and 2s, respectively. These values then index into the snow/slategray vector to select a color. We could accomplish the same thing by adding 1 to df$survived:

```
plot(df$age, pch=21, cex=0.75,
    bg=c("snow", "slategray")[df$survived+1], ylab="Age",
    main="Age (White Deceased)")
```

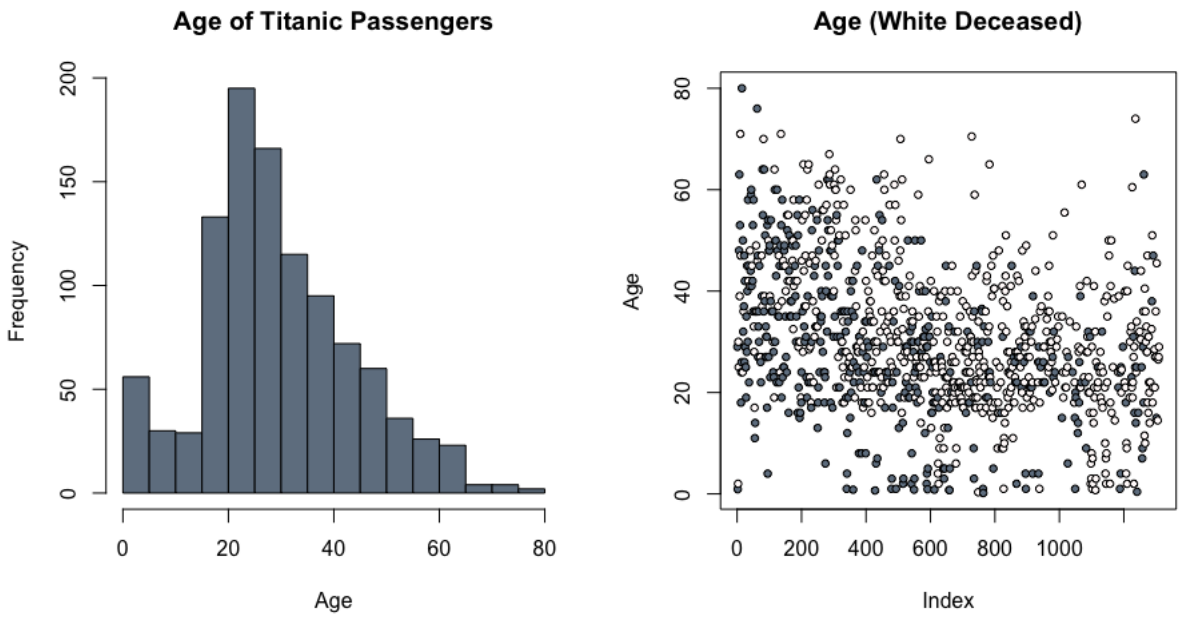


Figure 3.1: Plotting a Quantitative Vector

Another option for a quantitative vector is the kernel density plot. This plot gives you similar information as the histogram, but smoothing has been applied. In the code below, we first create the density vector, then use it for the plot() function. The last line of code below fills in the curve with a polygon.

Code 3.1.2 — Titanic data. Kernel Density Plot.

```
d <- density(df$age, na.rm = TRUE)
plot(d, main="Kernel Density Plot for Age", xlab="Age")
polygon(d, col="wheat", border="slategrey")
```

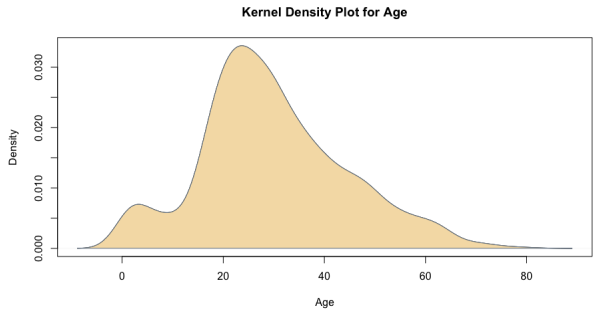


Figure 3.2: Kernel Density Plot

We can overlay several kernel density plots using package sm. First we subset the data frame to the two columns of interest so that we can use complete.cases() to get rid of NAs.

Code 3.1.3 — Age by Class. Overlaying Kernel Density Plots.

```
library(sm)
# subset the data and remove NAs
df_subset <- df[,c(1,5)]
df_subset <- df_subset[complete.cases(df_subset),]
# create the plots
sm.density.compare(df_subset$age, df_subset$pclass,
     col=c("seagreen", "wheat", "sienna3"), lwd=2)
title(main="Age by Passenger Class")
legend("topright", inset=0.05, legend=c(1:3),
     fill=c("seagreen", "wheat", "sienna3"))
```

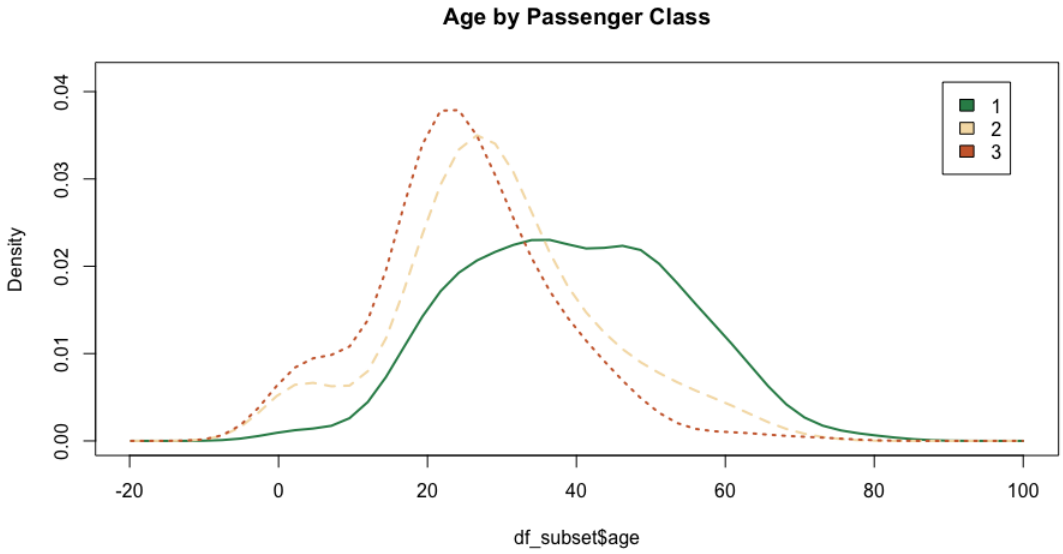


Figure 3.3: Kernel Density Plots

A boxplot is another graph type that can represent quantitative data. A box plot is more commonly vertical but below we show a horizontal example. The box shows the 2nd and 3rd quartiles of the data. The "whiskers" at either end of the dashed lines show the 1st and 4th quartiles. Dots beyond a whisker indicate suspected outliers. The outliner dots seem to be for ages greater than around 65. We know that these really aren't suspect data, since we have the human knowledge that people in the time of Titanic often lived at least until their 80s. The bold line through the box indicates the median. In this data, the median age appears to be in the late 20s.

Code 3.1.4 — Age Data. Horizontal Box Plot.

```
boxplot(df$age, col="slategray", horizontal=TRUE, xlab="Age",
    main="Age of Titanic Passengers")
```

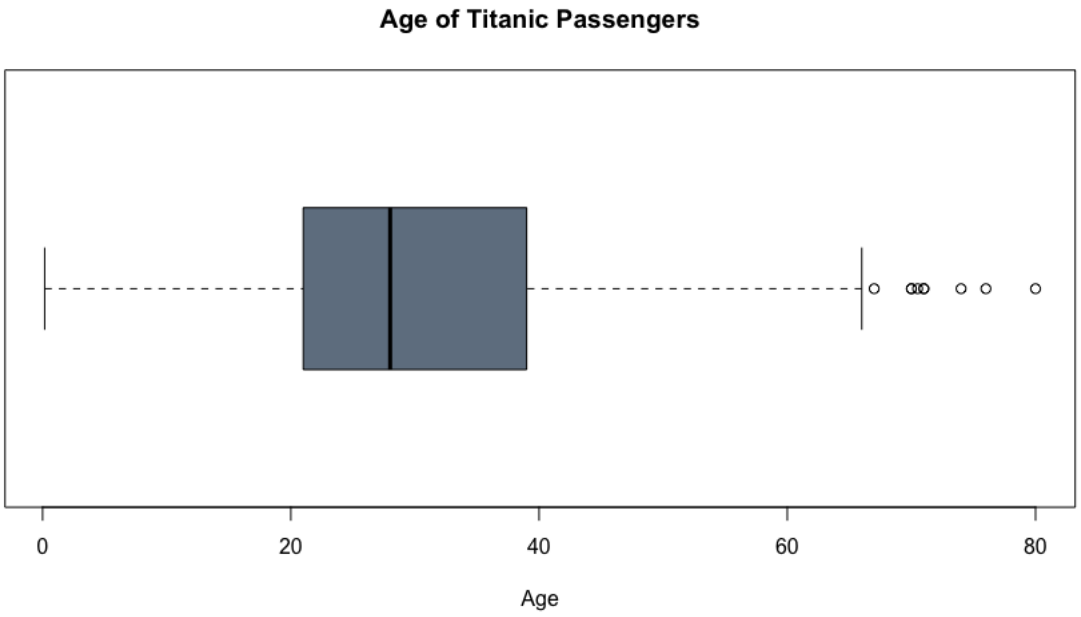


Figure 3.4: Box Plot

3.1.2 Plotting Qualitative Vectors

Barplots are often used for qualitative vectors. They can be vertical or horizontal. In the code below, adding parameter `horiz=TRUE` will cause the bars to be displayed horizontally instead of vertically. First, we make counts from the passenger class vector, then use those to create the bar plot.

Code 3.1.5 — Passenger Class Data. Bar Plot.

```
counts <- table(df$pclass)
barplot(counts, xlab="Passenger Class", ylab="Frequency",
     col=c("seagreen","wheat","sienna3"))
```

The table() function in the code block returns the following:

```
  1   2   3
323 277 709
```

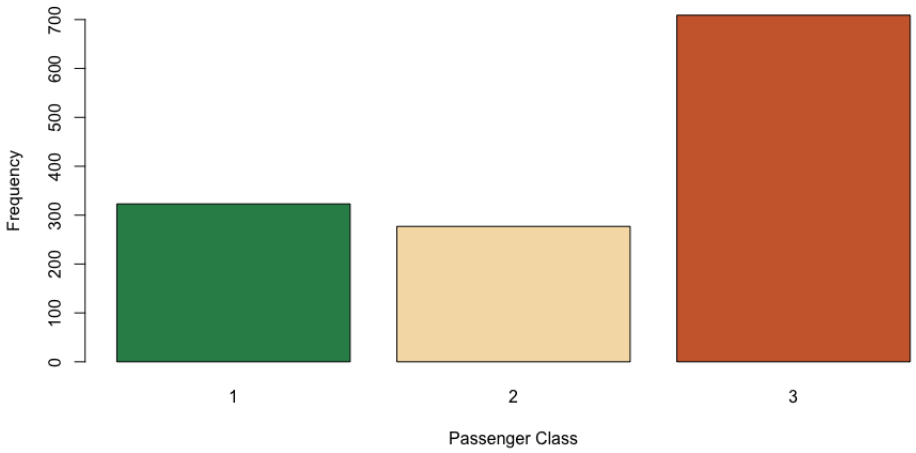


Figure 3.5: Bar Plot

A pie chart can be made with relative frequencies of a qualitative variable. First we specify frequencies for each of the 3 classes, then supply labels. With slices and labels defined, we can make a pie chart.

Code 3.1.6 — Passenger Class Data. Pie Chart.

```
slices <- c(sum(df$pclass==1, na.rm = TRUE), sum(df$pclass==2,
    na.rm = TRUE), sum(df$pclass==3, na.rm = TRUE))
lbls <- c("Class 1", "Class 2", "Class 3")
pie(slices, labels=lbls, main="Passenger Classes",
    col=c("seagreen","wheat","sienna3"))
```

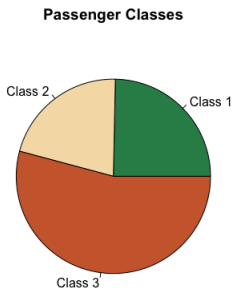


Figure 3.6: Pie Chart

3.2 Data Visualization of Two Vectors

If we have two vectors, X and Y, then there are four possible combinations of quantitative and qualitative vectors, listed below. In this section we look at graphs that are appropriate for each combination.

- both X and Y are qualitative
- X is qualitative, Y is quantitative
- X is quantitative, Y is qualitative
- X and Y are quantitative

3.2.1 Both X and Y are Qualitative

When both variables are qualitative, mosaic plots are the most common type of graph used. A related type is the association graph, which gives additional visual information about the deviation of the data from a uniform distribution. Both types of plots can be created with the `vcd` package, visualizing categorical data. First, we look at a mosaic example plotting the survived and pclass columns. The mosaic() function wants the first argument to be a table or formula, so we surround the subsetted data frame with table(). SHADE=TRUE gives you a color graph, FALSE gives you a grayscale graph. The mosaic plot shows each group in tiles. The area of the tiles is proportional to its counts in the data.

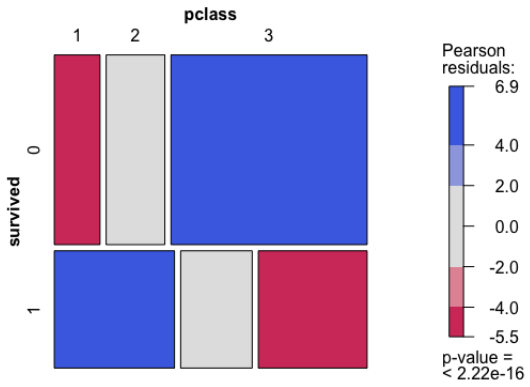


Figure 3.7: Mosaic Plot

The legend indicates the Pearson residuals. The "null" model would consider an even distribution into the cells but clearly we don't have that case here. The blue indicates we have more observations than expected, the red indicates fewer than expected, and gray is about what is expected given a null hypothesis. We didn't have to specify legend=TRUE because that is the default.

Code 3.2.1 — Passenger Class and Survived. Mosaic and Association Plots.

```
library(vcd)
mosaic(table(df[,c(2,1)]),  shade=TRUE, legend=TRUE)
assoc(table(df[,c(1,2)]), shade=TRUE)
```

What would happen if we reversed the order of columns 2 and 1? We would get the same information, but with the graph flipped around.

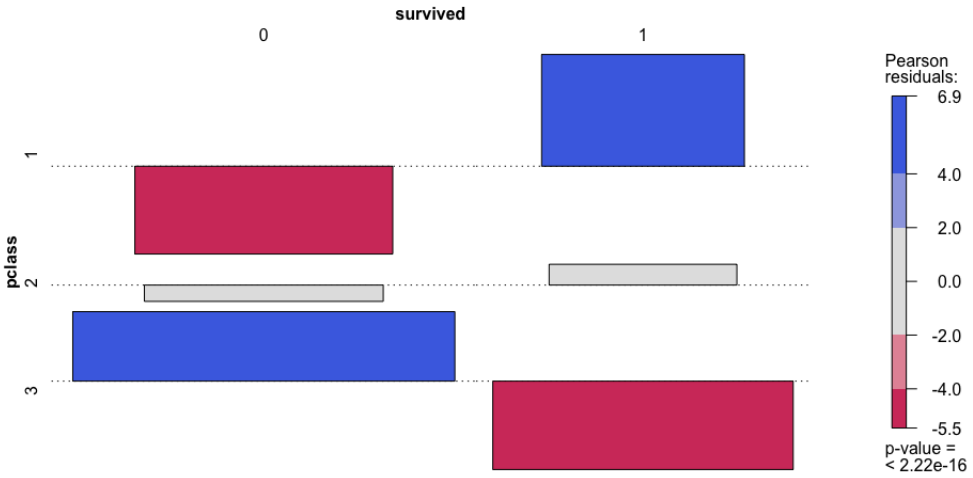


Figure 3.8: Association Plot

An association plot visualizes the residuals of an independence model. Each tile has an area that is proportional to the difference in observed and expected frequencies. The dotted line is the baseline. Tiles above the line have a frequency greater than what was expected, those below have a frequency below what was expected. In the plot above, pclass 1 survived more than expected, pclass 3 less than expected.

3.2.2 X is Qualitative, Y is Quantitative

When X is qualitative (a factor), and Y is quantitative, box plots are good choices. Notches at the median can be added with the notch=TRUE parameter.

Code 3.2.2 — Passenger Class and Age. Box Plot.

```
plot(df$survived, df$age, varwidth=TRUE, notch=TRUE,
    main="Survival and Age", xlab="Survived", ylab="Age")
# the following would create an identical plot
boxplot(df$age~df$survived, varwidth=TRUE, notch=TRUE,
    main="Survival and Age", xlab="Survived", ylab="Age")
```

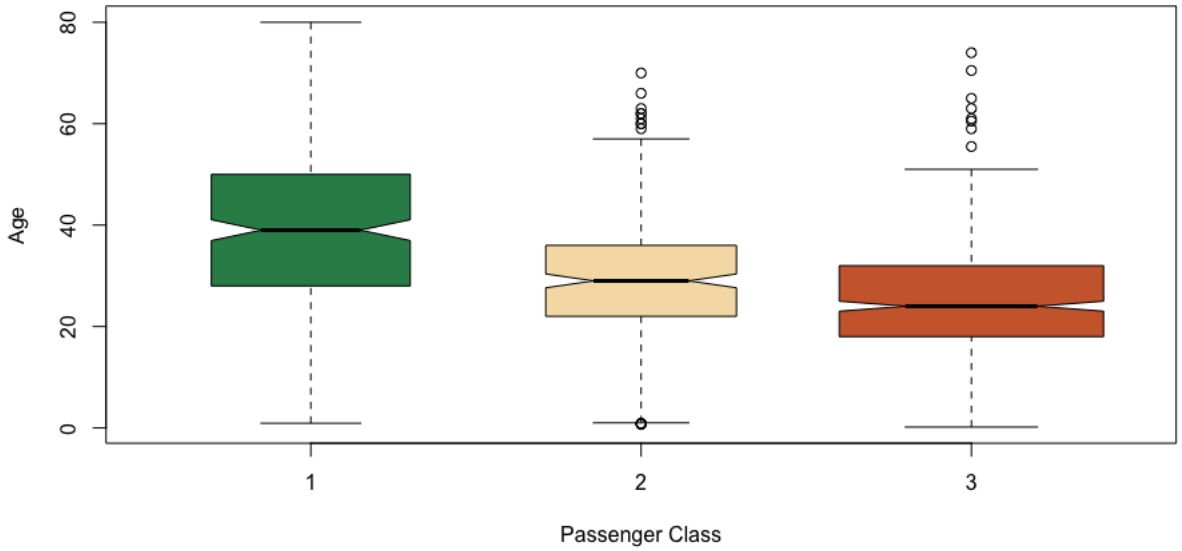


Figure 3.9: Box Plot

You can create violin plots with package vioplot. Violin plots are a combination of a boxplot and a kernel density plot. This plot does not like NAs so we remove them. This means that we will have less data and the plot may not be as informative.

Code 3.2.3 — Passenger Class and Age. Violin Plot.

```
library(vioplot)

# just look at columns 1, 2, 5
df_subset <- df[,c(1,2,5)]
# subset to remove NAs
df_subset <- df_subset[complete.cases(df_subset),]

x1 <- df_subset$age[df_subset$pclass==1]
x2 <- df_subset$age[df_subset$pclass==2]
x3 <- df_subset$age[df_subset$pclass==3]

# make the plot
vioplot(x1, x2, x3, col="wheat",
    names=c("Class 1", "Class 2", "Class 3"))
```

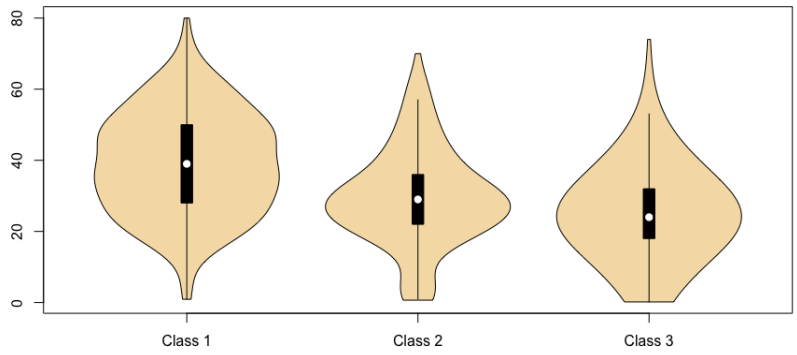


Figure 3.10: Violin Plot

3.2.3 X is Quantitative, Y is Qualitative

When X is quantitative and Y is qualitative, a conditional density plot can be used. The following plot shows how survived changes over the various ages. Note that if we switched the order of age and survived, we would get a row of dots at the top of the graph for one class and a row of dots at the bottom of the graph for the other class, not terribly informative.

Code 3.2.4 — Passenger Class and Age. Conditional Density Plot.

```
cdplot(df_subset$age, df_subset$survived, col=c("snow", "gray"))
```

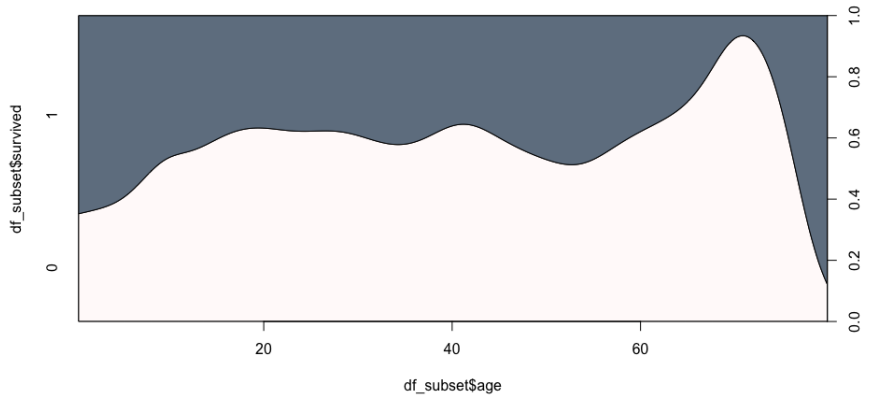


Figure 3.11: Conditional Density Plot

3.2.4 X and Y are both Quantitative

If X and Y are both quantitative, scatter plots are recommended. Here we have crosses for the points in blue, 75% of the usual size (by parameter cex). We would have to dig further into the Titanic data to understand this chart. Why do so many passengers seem to have a fare of 0? And why did a few passengers pay 500? Perhaps the 500 fares paid for several people and the 0 fares reflect passengers whose fares were paid by a spouse or parent or adult child? Further investigation is required to understand this.

Code 3.2.5 — Fare and Age. Scatter Plot.

```
plot(df$age, df$fare, pch='+', cex=0.75, col="blue",
    xlab="Age", ylab="Fare")
```

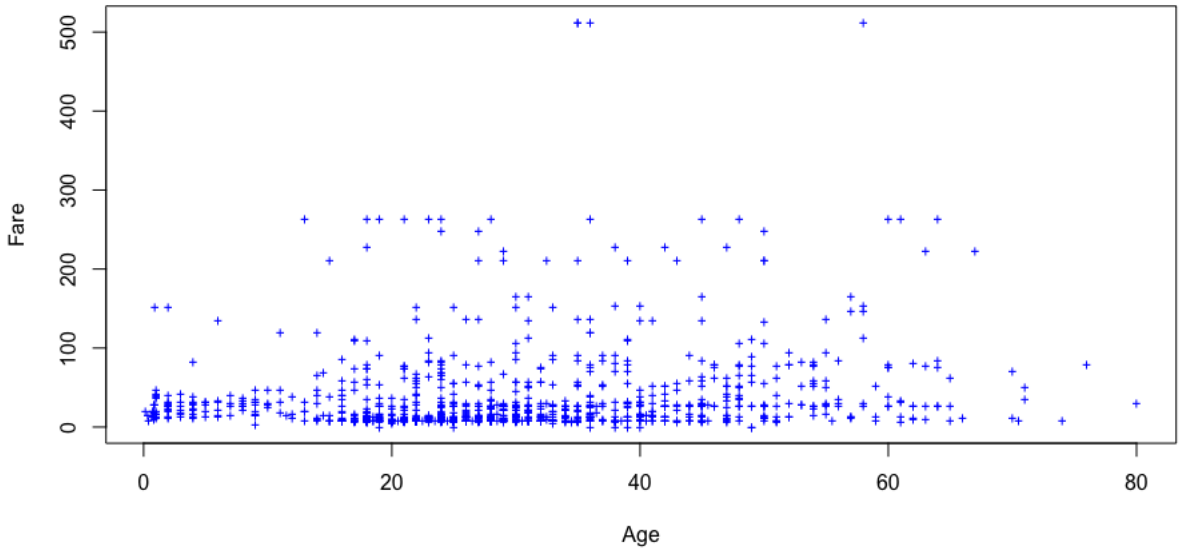


Figure 3.12: Scatter Plot

3.3 Summary

This chapter demonstrated how to create informative graphs in basic R. We have really just scratched the surface of what can be done in R. The data visualization capabilities of R are one of many reasons that it is heavily used in industry and academia. Data visualization is useful first for a researcher's understanding of the data, and then to communicate what has been learned about the data to others. There are many online resources for advanced data visualization techniques in R. Advanced graphics can be achieved with the power ggplot2 package, described in a later chapter.

3.3.1 Quick Reference

Reference 3.3.1 Change and restore graphing parameters

```
opar <- par()        # copy original settings
par(mfrow=c(1,2))   # set up 1x2 grid
hist(...    # make a plot
plot(...   # make another plot
par(opar)   # restore parameter settings
```

3.3.2 R Plot Parameters

The next few code chunks show how to change a few plot paramters.

Reference 3.3.2 Text and symbol size

```
# cex changes point size
# default is 1
# 1.5 means 50% larger
plot(mtcars$wt, mtcars$mpg, cex=1.5)
```

Reference 3.3.3 Point symbol

```
# pch is point symbol; default is open dot
# pch=19 is closed dot, filled with col

plot(mtcars$wt, mtcars$mpg,
      pch=19, col='green')
```

Reference 3.3.4 lines

```
# lty 5-long dash; 2=short dash; 3=dots
# pch=19 is closed dot, filled with col

plot(mtcars$wt, mtcars$mpg)
abline(lm(mtcars$mpg~mtcars$wt), lty=5, col='red')
```

3.3.3 Practice to Consolidate Skills

This chapter introduced you to R, often called Base R or Standard R. This is the R language that has been in use since the 1980s. The importance of learning base R is to be able to read and use legacy code. Base R is all you really *have* to know to use R for machine learning. However, your skill set is greatly enhanced by learning new additions to R syntax, Modern R. This is the purpose of the next chapter.

One of the enhancements in Modern R is ggplot2, which is a more powerful data visualization packages. However, most of the time when I am exploring my data I use Standard R because it is faster and simpler to just look at my data compared to ggplot2, and I love faster and simpler.

Problem 3.1 — Data Exploration 1. In this exercise you will explore a built-in data set in R, using techniques covered in Chapters 2 and 3, and beyond.

1. Create a new R Notebook (Rmd file) and save it to your computer. Create relevant headings and commentary throughout.
2. At the console, type 'data()' to open a new tab to see all the data sets available in base R.
3. In a code block, use summary() and str()
4. At the console type: ?CO2 and choose the first option if there is more than one. Read the documentation that should pop up on the lower right.
5. Below your code block, write a brief summary of the data set, including a description of the purpose of the data, and the type of data in each column.
6. Check if the data set has any NAs.
7. In a 1x2 grid, plot **Type** and **conc**, and plot **Type** and **uptake**. Label the y axis for each plot.
8. The Quebec uptake plot in the previous step showed some outliers. Take a closer look by plotting a histogram. Then perform a summary() on that one column and compare the histogram with the summary. Write your insights into this column.
9. In a 1x2 grid, plot **Treatment** and **conc**, and plot **Treatment** and **uptake**. Label the y axis for each plot.
10. The first column is an ordinal factor column. Research what this means and when it might be used. Write a one to two sentence summary of what you learned.
11. Subset the data (into a new data frame) to just Qn1, Qn2, and Qn3. Then plot again. What went wrong?
12. We don't want to see any other plants than the ones in our subset. Research R's droplevels() function, then drop unused levels and plot again.

Problem 3.2 — Data Exploration 2. In this exercise we will do some data exploration on the Penguin data set from Kaggle: `https://www.kaggle.com/code/parulpandey/penguin-dataset-the-new-iris`. Download the data to your computer. First, create a new R rmd file and create relevant headings and commentary throughout.

1. Read in the csv rile, using stringsAsFactors=TRUE. You can do one of 3 things to allow R to find the file: (1) make sure the csv file and your notebook are in the same directory, (2) use R function setwd() to set the working directory to wherever you saved the file, or (3) go to File, then Import Data set to load it into R.
2. There are 3 levels for factor **sex**. Use the unique() function to figure out why.
3. Get rid of rows where sex is ".", then recode. There are various ways to do this, but it may make it easier to convert df$sex to character first, then subset, then refactor. Use the str() function on the updated data set to make sure all is well.
4. Create at least 3 plots of the data to learn more about the data and write a brief narrative about what you learned.

3.3.4 Next-Level Learning

This exercise is best completed with a learning partner.

Problem 3.3 — Data Exploration 3. In this exercise you will pick another data set and learn to ask your own questions about data.

- Create a new R Notebook (Rmd file) and save it to your computer. Create relevant headings and commentary throughout.
- Find a data set that is new to you. Explore the data set using R functions. Write a short narrative description of the data.
- In your notebook, create 5 or more sets of instructions and R code block implementations that explore the data.
- If you are working with a partner, blank out the code in your R blocks, they will do the same, then exchange notebooks so you can each practice coding.

4. Modern R

Chapters 2 and 3 (hopefully) made you comfortable with coding in standard R. This chapter introduces you to extensions to the language that fall under the general title of the **tidyverse**. The tidyverse is a set of R packages designed for improved data manipulation, exploration, and visualization. The packages work together well because they have common data representations and a common API design. See the recommended method to install the tidyverse on your machine here: `https://www.tidyverse.org/` and also see the *R for Data Science* free online book, which takes a very deep dive into R. In the interests of getting to the machine learning quickly, this chapter gives you a sweeping tour of the tidyverse packages. When you run `library(tidyverse)`, R will load the following packages:

- dplyr - provides a grammar of data manipulation
- tibble - creates and manipulates a modern data frame
- tidyr - helps you create tidy data
- readr - is a fast reader of rectangular data
- purr - enhances R's functional programmming (FP) toolkit
- ggplot2 - implements a grammar of graphics for data visualization
- stringr - makes working with strings easier
- forcats - work with factors
- libridate - for dates and times

These extensions to the language are exciting because they have updated R to be more competitive with newer languages. The Base R language was developed in the 1980s when data was smaller and computing power and memory were limited compared to today. In comparison, the tidyverse is fast, and better able to handle larger data. This chapter first explores tidyverse syntax for data manipulation and then looks at ggplot2 for creating graphs.

A key concept of the tidyverse is that data should be tidy, that is organized into rows of observations and columns of variables as in a data frame. Each value represents one observation's value for one variable. This may seem obvious because the data sets we have worked with in R, in R packages, and from sites like Kaggle are already tidy. However, real-world data is often a mess because it was organized years, perhaps decades ago, to facilitate data entry. The tidyverse contains multiple functions to take a pile of messy data and shape it into tidy data. That will not be our focus in this volume because we want to focus on machine learning once data is tidy. However, keep in mind that if you find yourself in a situation where you are hired to clean up someone's data you should check out the tidyverse functions before writing custom code.

4.1 Tibble

A *tibble* is the tidyverse update to the Standard R data frame. A tibble is backwardly compatible with the data frame, so you can use a tibble anywhere you previously used a data frame. A tibble is the central data structure for the tidyverse. The documentation states that the general ethos of tibbles is that tibbles are lazy, because they do less, and surly, because they complain more to alert you to problems in the data.

A tibble behaves a little differently than a data frame under the hood but most of the implementation details will not be noticeable to the average user. Two differences you will notice are:

- A tibble will print only a screen at a time of the data.
- A subset of a tibble always returns a tibble. Subsetting a data frame could return a data frame or just a vector.

The code below loads a data frame into memory and converts it to a tibble with the tbl_df() function.

Code 4.1.1 — tibble. Create a tibble.

```
library(tidyverse)
library(mlbench)

data("PimaIndiansDiabetes2")   # load the data into memory

tb <- as_tibble(PimaIndiansDiabetes2)  # convert df to tibble
rm(PimaIndiansDiabetes2)  # remove df from memory
tb # display the tibble
```

```
# A tibble: 768 x 9
```

```
   pregnant glucose pressure triceps insulin  mass pedigree   age diabetes
      <dbl>   <dbl>    <dbl>   <dbl>   <dbl> <dbl>    <dbl> <dbl> <fct>
 1        6     148       72      35      NA  33.6    0.627    50 pos
 2        1      85       66      29      NA  26.6    0.351    31 neg
 3        8     183       64      NA      NA  23.3    0.672    32 pos
 4        1      89       66      23      94  28.1    0.167    21 neg
 5        0     137       40      35     168  43.1    2.29     33 pos
 6        5     116       74      NA      NA  25.6    0.201    30 neg
# ...
```

A slightly different view is available with the `glimpse(tb)` function. The output is similar to that of str() in Base R.

```
>glimpse(tb)

Observations: 768
Variables: 9
$ pregnant <dbl> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, . . .
$ glucose  <dbl> 148, 85, 183, 89, 137, 116, . . .
$ pressure <dbl> 72, 66, 64, 66, 40, 74, 50, . . .
$ triceps  <dbl> 35, 29, NA, 23, 35, NA, 32, . . .
$ insulin  <dbl> NA, NA, NA, 94, 168, NA, 88, . . .
$ mass     <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, . . .
$ pedigree <dbl> 0.627, 0.351, 0.672, 0.167,. . .
$ age      <dbl> 50, 31, 32, 21, 33, 30, 26, 29, . . .
$ diabetes <fct> pos, neg, pos, neg, pos, neg, . . .
```

4.2 Pipes

The pipe symbol, %>%, works like the unix pipe. It makes code easier to read as you can combine several commands in a neat group of lines instead of nesting functions in the typical R fashion. The pipe symbol can be typed as you see it, or you can use the shortcut shift+command+M. Pipes are most useful when performing a series of data manipulations on a tibble, but can be used in other places as well.

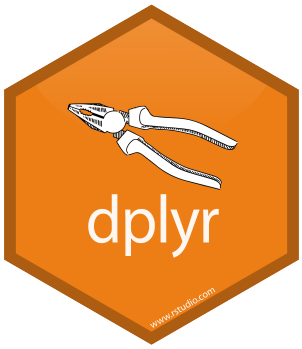
It is important to be able to read the two different syntax styles or personalities of R. For example, here is how we might compute root mean squared error in standard R:

```
rmse <- sqrt(mean((pred - test)^2))
```

To understand the code above you have to start from the inner ( ) and work outward. That is, perform vector subtraction of test values from the predicted valuesIn pipes, you just read down:

```
# compute rmse with pipes
rmse <- (pred - test)^2  %>%
     mean   %>%
     sqrt
```

4.3 Package dplyr



According to the package author, Hadley Wickham, the d in dplyr is for data and the rest is to evoke pliers. So dplyr is used to manipulate data efficiently. The dplyr packages has several functions for data manipulation. Some of the functions work on columns, while others work on rows. The dplyr package is considered to be a grammar of data manipulation, so notice that all the function names are verbs. The dply library has dozens of functions, see the documentation here: `https://dplyr.tidyverse.org/reference/index.html` This section will focus on some of the most commonly used.

Functions that work on columns:

- glimpse - similar to standard R's str() structure function
- select - subset columns
- rename - rename columns
- mutate - create, modify, or delete columns

Functions that work on rows:

- filter - subset rows that match conditions
- arrange - order rows based on column values
- distinct - keep unique rows

Functions that work on groups of rows:

- group_by - group by one or more variables
- summarize() or summarise() - somewhat similar to summary() in standard R, notice either American or British spelling works the same

4.3.1 Select

The select function selects columns, returning a tibble with just those columns. The following code uses select() to extract two columns from the data frame and send them to the print function. Although select() returns a new tibble, in this case we didn't save it to another variable or back to itself so it does not exist in memory.

Code 4.3.1 — dplyr. Select.

```
select(tb, diabetes, pregnant) %>%
   glimpse

   Observations: 768
Variables: 2
$ diabetes <fct> pos, neg, pos, neg, pos, neg, pos, neg, ...
$ pregnant <dbl> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, 10, 10, ...
```

4.3.2 Mutate

You can use mutate to create new columns from the existing data. In the code example below we created a new binary factor columns that is 1 if glucose is above average for the population and 0 otherwise. The mutate() function returns a tibble, the code below replaces the tibble. Notice that you can use square brackets to subset a tibble just as you can for data frames.

Code 4.3.2 — dplyr. Mutate.

```
tb <- tb %>%
     mutate(glucose_high = factor(
         ifelse(glucose>mean(glucose, na.rm=TRUE), 1, 0)))

tb[1:5, c(2, 10)]  # display glucose and glucose_high for 5 rows

glucose   glucose_high
<dbl>    <fctr>
   148       1
    85       0
   183       1
    89       0
   137       1
```

Mutate can also be used to delete a column by setting it to null. The following code would remove the column created in the previous code:

```
tb <- tb %>%
    mutate(glucose_high = NULL)

names(tb)  # verify that the column is gone
[1] "pregnant" "glucose"  "pressure" "triceps"  "insulin"  "mass"
     "pedigree" "age"      "diabetes"
```

4.3.3 Rename

Rename a column using syntax: new = old.

```
tb <- rename(tb, blood_pressure = pressure)
```

There is also a handy `rename_with` method to rename multiple columns with a function. This is handy when you need to rename multiple columns in a similar fashion, as in modifying them all to uppercase or some customized pattern.

The previous methods `select, mutate, glimpse, rename` operated on columns. Next we look at methods that operate on rows.

4.3.4 Filter

The filter function is used to remove rows. Below we filter rows out that have NAs in either glucose or mass, then glimpse the data. The data now has 752 rows instead of 768.

Code 4.3.3 — dplyr. Filter.

```
tb <- filter(tb, !is.na(glucose), !is.na(mass))
dim(tb)

[1] 752 9
```

4.3.5 Arrange

Arrange is used to control the order of rows. Here we arrange by mass in descending order.

Code 4.3.4 — dplyr. Arrange.

```
arrange(tb, desc(mass)) # descending order
```

Next we look at dplyr methods that operate on the entire tibble, namely summarize() and group_by().

4.3.6 Summarize

The summarize function is a powerful way to get summary statistics. Below we have two examples. Note that you can spell it the American or British way: summarize or summarise. The syntax is: label=function, so that the example below displays a tibble with 3 columns min, max, and sd defined by the corresponding functions. This tibble is displayed, not stored in memory.

```
tb %>%
  summarize(min=min(mass), max=max(mass), sd=sd(mass))

  min      max      sd
  18.2   67.1    6.928926
```

The next example displays a tibble with two columns indicating the number of observations with positive diabetes and the number of observations that are negative for diabetes.

```
tb %>%
  summarize(num_diabetic = sum(diabetes=="pos"),
        num_healthy = sum(diabetes=="neg"))

  num_diabetic   num_healthy
<int>          <int>
264              488
```

An alternate way to get the same information would be to use the count() method:

```
tb %>% count(diabetes, sort = TRUE)

diabetes      n
<fctr>        <int>

neg           488
pos           264
```

4.3.7 Grouping

The group_by provides a way to summarize selected subsets of the data.

```
tb %>%
  group_by(diabetes) %>%
  summarize(median_BMI = median(mass, na.rm=TRUE))

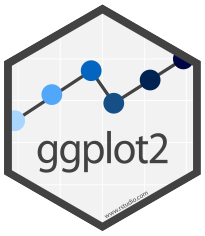
diabetes    median_BMI
<fctr>      <dbl>

neg         30.10
pos         34.25
```

Other dplyr methods that may be useful include methods to combine multiple data frames by rows or by columns, including complex joins, intersects, unions and more that will take you back to your database class. There are also built-in data sets you can play with, including starwars and storms. The tidyverse has updated some standard R functions for efficiency and power, but politely gave them different names so you can still use the superseded methods. For example, sample_n() can be used instead of the trusty sample() and top() can be used in place of head() or tail(). This will be covered more in the next chapter on data wrangling with R.

4.4 ggplot2

Hadley Wickham developed ggplot2 in 2005, inspired by a grammar of graphics developed by Leland Wildinson in 1999. The ggplot2 functions are much more powerful than standard R graphs but also slower.



Some R users prefer to use base R plots for data exploration, the plots that only they will see that give insight into the data. Then later they can spend the time producing a pretty showpiece graph for publication.

The gglot2 package can be loaded with the library() function but is also loaded automatically when the tidyverse is loaded: `library(tidyverse)`

There are 7 grammatical elements in ggplot2, the first 3 of these are essential to getting something plotted.

- data = the data being plotted should be the first argument, or specify data=...
- aesthetics - the scales onto which we plot; use aes() to specify at least x= and y= if needed as well as other parameters for customization
- geometries - visual elements such as points, lines, etc.
- facets - for plotting multiples
- statistics - representations to aid understanding
- coordinates - space on which data will be plotted
- themes - you can customize your own theme to use over and over

To get started, the code below shows the important components of building a ggplot:

- the data
- the aesthetics which are how the data is represented
- the geometry
- labels

The full code is available in the GitHub. The graph is visible below. As you can see, the graph is a little blah. A little color would be nice. Also, the points seem to be a blob and it's hard to find a trend. We can add a smoothing line to show that as BMI increases, glucose increases. The code is shown in the second code block below, with the graph underneath.

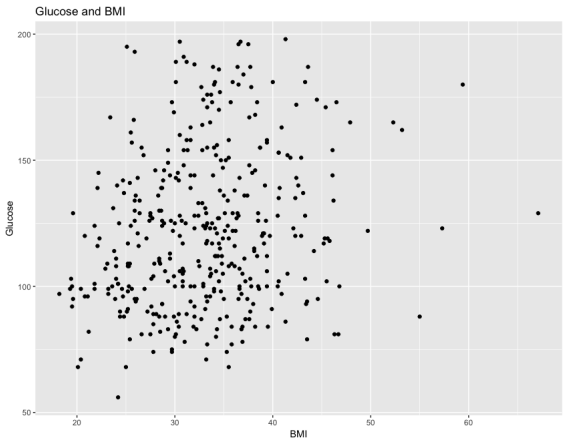


Figure 4.1: Scatterplot of BMI and Glucose

Code 4.4.1 — ggplot. Basic Scatterplot Using PimaIndiansDiabetes2 data.

```
ggplot(df, aes(x=mass, y=glucose)) +
  geom_point() +
  labs(title="Glucose and BMI", x="BMI", y="Glucose")
```

Code 4.4.2 — ggplot. Scatterplot and Smoothing Line.

```
ggplot(df, aes(x=mass, y=glucose)) +
  geom_point(pch=20, color='blue', size=1.5) +
  geom_smooth(method='lm', color='red', linetype=2) +
  labs(title="Glucose and BMI", x="BMI", y="Glucose")
```

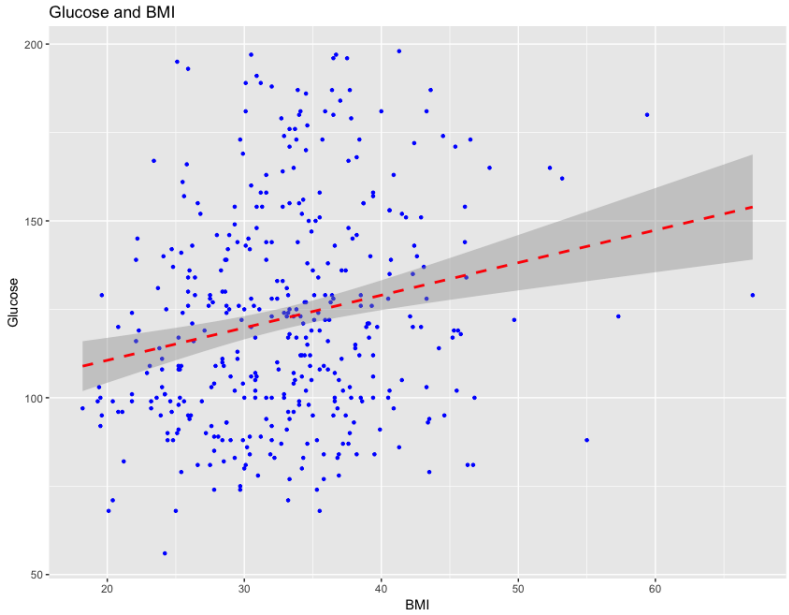


Figure 4.2: Scatterplot with Smoothing Line

4.4.1 Facet Grid

In the notebook on the GitHub repo, we made two new factor columns, glucose_high and insulin_high which are 1 if the value is above the mean and 0 otherwise. Then we plot with mass on the x axis and age on the y axis. We have 4 plots within the space based on the two new factor columns. Also the points are shaped according to whether the observation has diabetes or not and the color indicates the number of pregnancies.

Code 4.4.3 — ggplot. Facet Grid.

```
ggplot(df,
  aes(x=df$mass, y=df$age, shape=diabetes, col=pregnant)) +
  geom_point(size=2) +
  facet_grid(df$glucose_high~df$insulin_high)
```

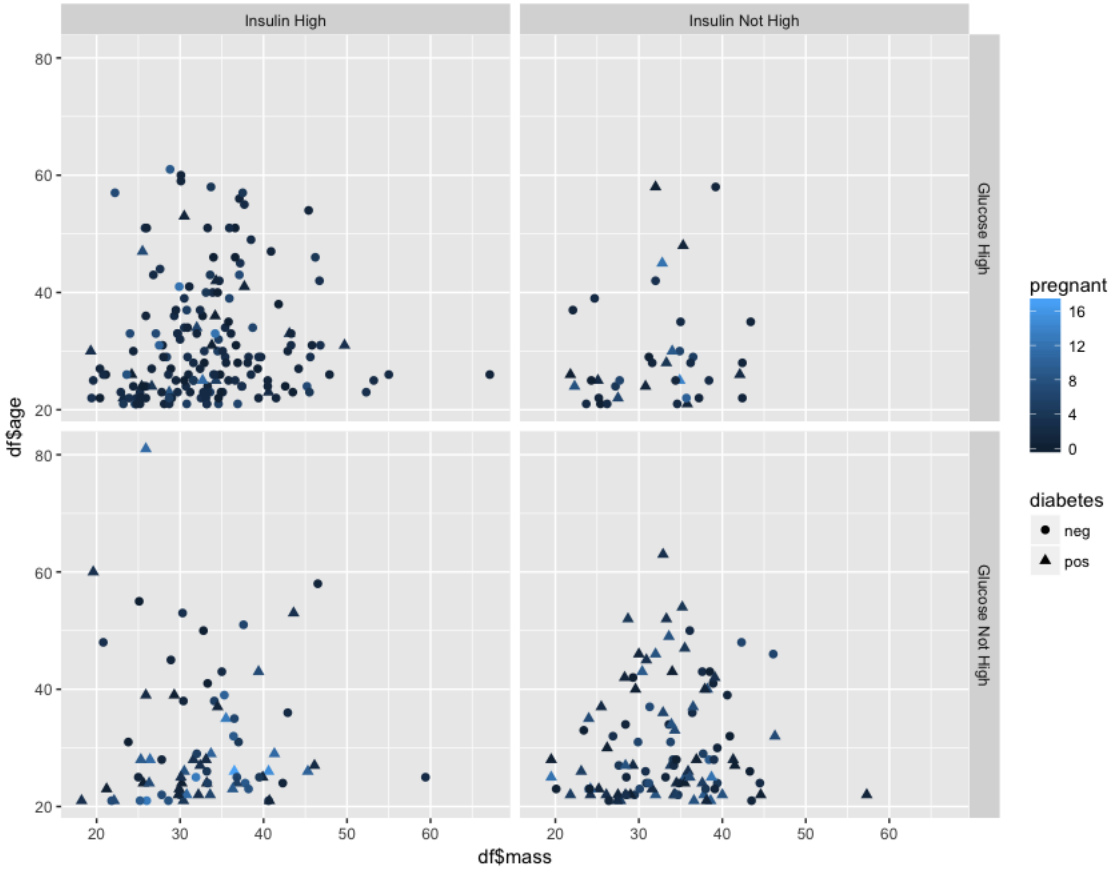


Figure 4.3: Facet Grid

4.4.2 Histogram

With standard R we can create a histogram with `hist(df)` if we just want to see the data. This is true for other types of graphs such as the boxplot. A histogram with ggplot requires a bit more syntax.

Code 4.4.4 — ggplot. Histogram.

```
ggplot(df, aes(x=mass)) +
  theme_bw() +  # comment this line out for default gray theme
  geom_histogram(fill="cornsilk4")
```

4.4.3 Overwhelmed?

Consulting the documentation for ggplot can be a bit overwhelming: `https://ggplot2.tidyverse.org/reference/`. This is why we are demonstrating some common graph types here. The cheatsheat in the GitHub repo is helpful for discovering more types of graphs, and a helpful mapping of data types to graph types.

With ggplot you could make your own distinctive theme to give your graphs a custom look. So far this chapter has used the default theme, but it can be changed as we see in the code above with the line `theme_bw()`.

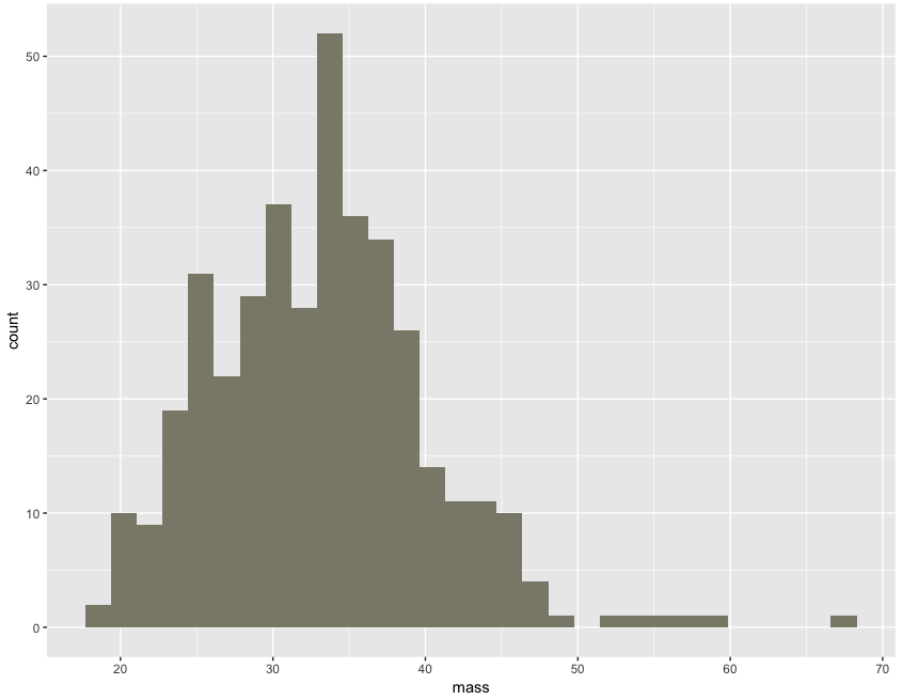


Figure 4.4: Histogram

4.4.4 Boxplot and Rug

A Boxplot is shown in Figure 4.5. The rug on the side shows the distribution. The points overlaying the boxplot add more info. The boxes are notched at the median. Here is the code:

Code 4.4.5 — ggplot. Boxplot and Rug.

```
ggplot(df, aes(x=diabetes, y=mass)) +
  geom_boxplot(notch=TRUE) +
  geom_point(position="jitter", color="cornflowerblue", alpha=.5) +
  geom_rug(color="cornflowerblue")
```

The ability of ggplot to add layers of information to graphs makes them much more informative compared to more simple plots in standard R.

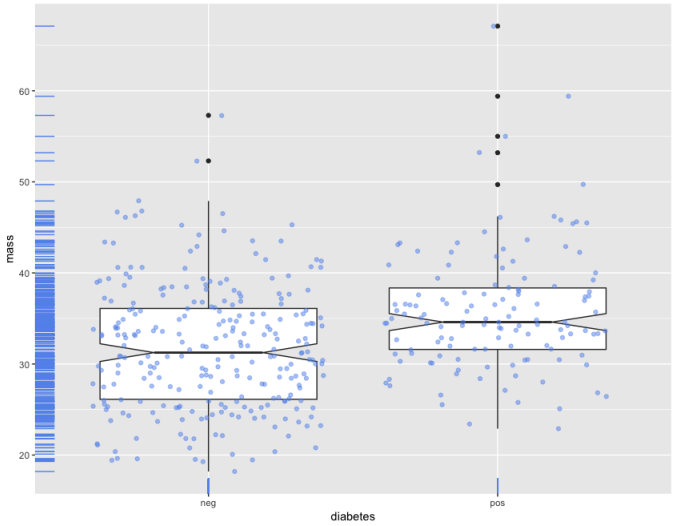


Figure 4.5: Boxplot and Rug

4.4.5 Density Plot

This plot lets you see the density of diabetes positive and negative overlapping each other because the alpha parameter makes them semi translucent. See Figure 4.6.

Code 4.4.6 — ggplot. Density Plot.

```
ggplot(df, aes(x=mass, fill=diabetes)) +
  geom_density(alpha=0.4)
```

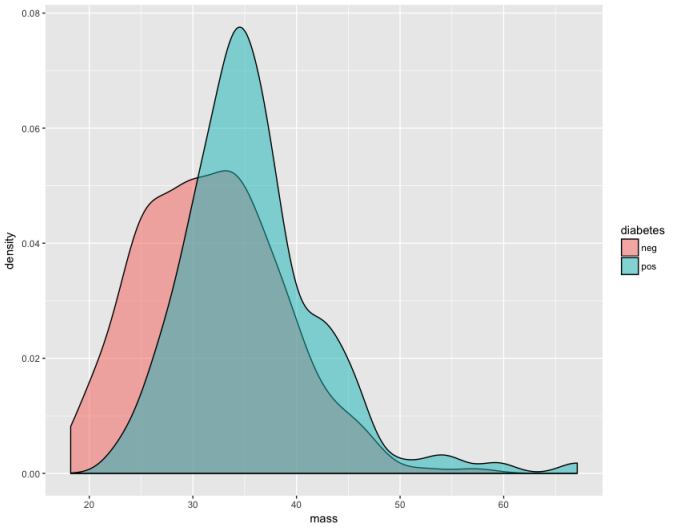


Figure 4.6: Density Plot

4.4.6 Bubble Chart

The bubble size is indicative of the number of pregnancies. The `size=pregnant` argument makes the size of shape 21 vary for each point according to the magnitude of the pregnant collumn. See Figure 4.7.

Code 4.4.7 — ggplot. Bubble Chart.

```
ggplot(df,
       aes(x=mass, y=glucose, size=pregnant)) +
  geom_point(shape=21, fill="cornflowerblue")
```

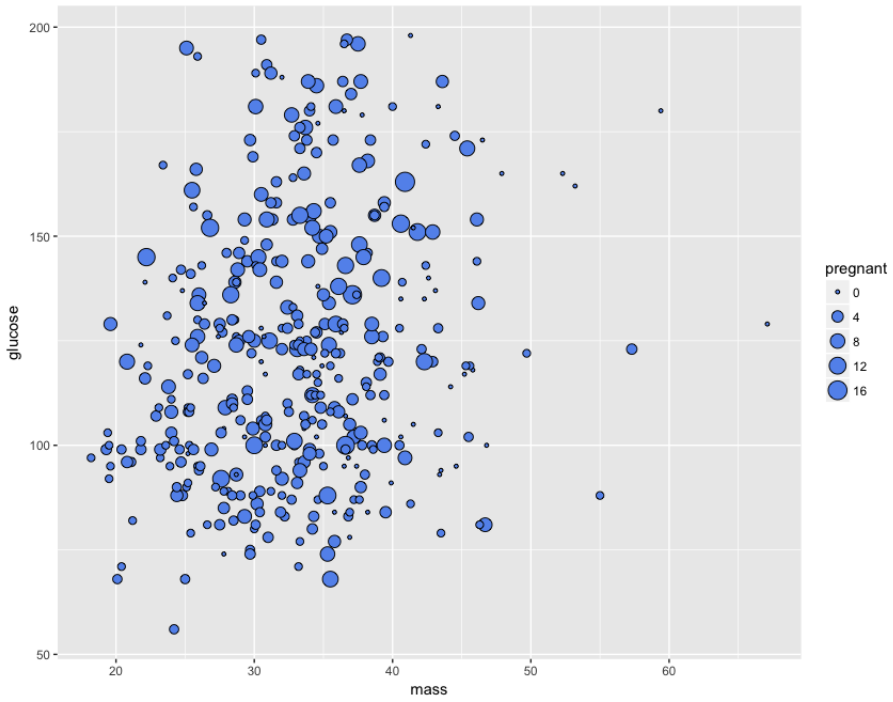


Figure 4.7: Bubble Chart

4.4.7 Grid

In standard R we arrange plots in a grid layout with par(). In ggplot we use grid.arrange().

Code 4.4.8 — ggplot. Grid Arrange.

```
library(gridExtra)
p1 <- ggplot(df, aes(x=insulin_high)) + geom_bar(fill="blue")
p2 <- ggplot(df, aes(x=glucose_high)) + geom_bar(fill="blue")
grid.arrange(p1, p2, ncol=2)
```

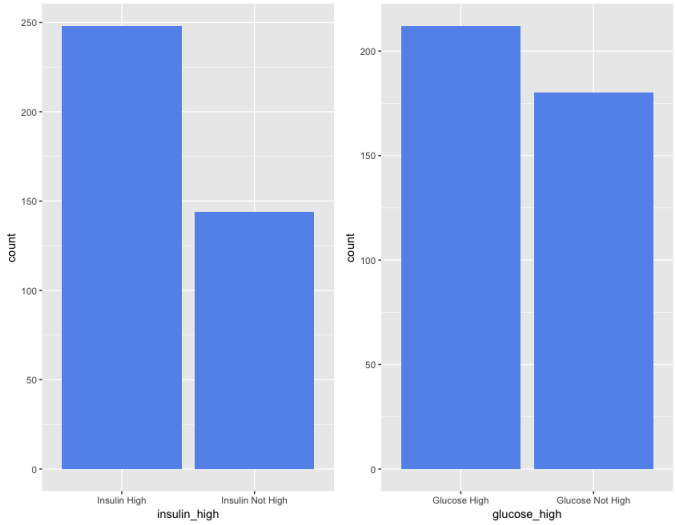


Figure 4.8: Grid Arrange

4.5 Summary

The name **ggplot2** invokes a grammar of graphics. Plots are specified layer by layer, with the minimum layers needed are data, aesthetics (what you see as in x y variables), and geometric objects like lines, points, bars.

Compared to standard R, plotting with ggplot2 is more complex but the result can be visually stunning. This chapter has given you the tools you need to get started making your own plots. Whether you choose to communicate data simply or to craft an arresting showpiece is a matter of personal preference, use case, and often time.

- A cheat sheet for ggplot2 from RStudio: `https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf`
- Colors in R can be referenced numerically or by name. Here is a list of names: R colors
- Quick-R reference for things like plotting symbols in base R and ggplot: `https://www.statmethods.net/advgraphs/parameters.html`

4.5.1 Practice to Consolidate Skills

We can only learn by doing, so to get more confident making plots with ggplot2, iterate over the following exercise as many times as you can, with different data sets, and different graph types, bells and whistles.

Problem 4.1 — Becoming Proficient. In a new R rmd notebook with relevant headings and commentary throughout:

- Find a data set that interests you
- Create at least 5 graphs that explore the data
- Write a brief narrative of what you learned about the data

4.5.2 Next-Level Learning

Learning a new platform can often take the form of a spiral staircase. You take a few steps, then you find you have a higher view of what you were doing than before. Repeat this process.

Problem 4.2 — Revisit ggplot2. To help you take the next few steps up that spiral staircase, read through this post on Kaggle: `https://www.kaggle.com/code/rtatman/visualizing-data-with-g`

Chapters 2, 3 and 4 have given you enough information to do actual work in R, but we have just scratched the surface of this amazing language. We need to move on to machine learning, but you can explore R in depth in this free book: `https://r4ds.hadley.nz/`

5. The Craft: Planning to Learn

Learning any craft is a long-term commitment. Key points to keep in mind are:

- Be systematic. Develop an organized approach to learning. Set aside dedicated time each week for learning. Keep a small physical notebook by your workspace to document what you worked on. This will help you be accountable to yourself.
- Document your learning. Take notes on what you learn as you learn it. This will help cement your learning and will be an invaluable resource as your learning progresses. You can document what you learned in physical or electronic notebooks. In this volume we will be creating R notebooks. A good start with electronic organization is to download the GitHub notebooks for this volume.
- Find mentors. Mentors may be people in your company/organization with skills you wish to learn. However, you can also find mentors online by identifying people who demonstrate the kind of projects you want to create, and following them online.

It is important that you develop an organized and systematic work flow for your projects. You also need to develop a plan for ongoing improvement. Specific goals with timelines can be helpful. Sharing your work with others is highly recommended. Forming an in-person or online study group will accelerate your learning, since no one person can have complete insight into every problem.

5.0.1 Machine Learning Workflow

A systematic approach to a machine learning project is important for two reasons. First, once you set up a work flow, the process becomes almost automatic. Second, by having a systematic workflow, others (or you) can reproduce your results. The concept of *reproducible research* has gained importance in recent years to help researchers distinguish between innovation and

hype. Also, when you go back to your own work months from now, you want to quickly see what you did and how you did it without having to reinvent your work.

Typical machine learning projects follow a work flow similar to Figure 5.1. In an organizational setting, the project objectives, accuracy goals and final report involve interactions with project managers (shown in green rounded, shadowed rectangles with diagonal sketch lines in the work flow diagram).

The yellow squared rectangles involve data gathering, cleaning, and exploration. Data must be organized, cleaned, and processed in order to be useful. In this phase it may be discovered that data is incomplete or inadequate for the project objectives in which case a plan for further data collection must be developed. The data collection phase may take months to complete in some cases.

After data is ready for consumption, data exploration can take place. Initial data exploration can identify trends in the data that will be useful in selecting algorithms and features.

The blue boxes (rounded corners, no diagonal sketch lines) involve the machine learning: training, testing, and evaluating. As shown in the diagram, machine learning projects are often iterative, going through model variations until acceptable results are achieved on the validation data. In reality, the iteration could involve earlier steps if project objectives and goals need to be reconsidered in light of project results.

5.0.2 The Life Cycle of a Machine Learning Project

A large machine learning project for clients involves much more than simply running data through an algorithm. The needs of various stakeholders in a project must be considered. Stakeholders in a machine learning project include, at a minimum:

1. The clients for whom the project is being prepared, who are also typically the funders of the project. These clients could be internal managers of a data science firm, or external clients from organizations and businesses who wish to obtain information that will help the entity achieve organizational objectives.
2. The technical experts and domain knowledge experts that plan and carry out the project. A wide range of expertise may be needed in larger projects. These experts could include statisticians, computer scientists, data scientists, and others.
3. Any persons whose data was collected or who may be impacted by the project.

Most projects are not one and done. Once clients/managers start using the project, they will think of additional features that would be nice to have. Further, some things may not work as expected and revisions and updates need to be made. Some projects are short-lived and others just won't die. My first professional coding experience was in the 1980s in COBOL. It horrifies me to think of all the COBOL code that is still out there in banks, government agencies, and more. Be kind as your create your projects. Document!

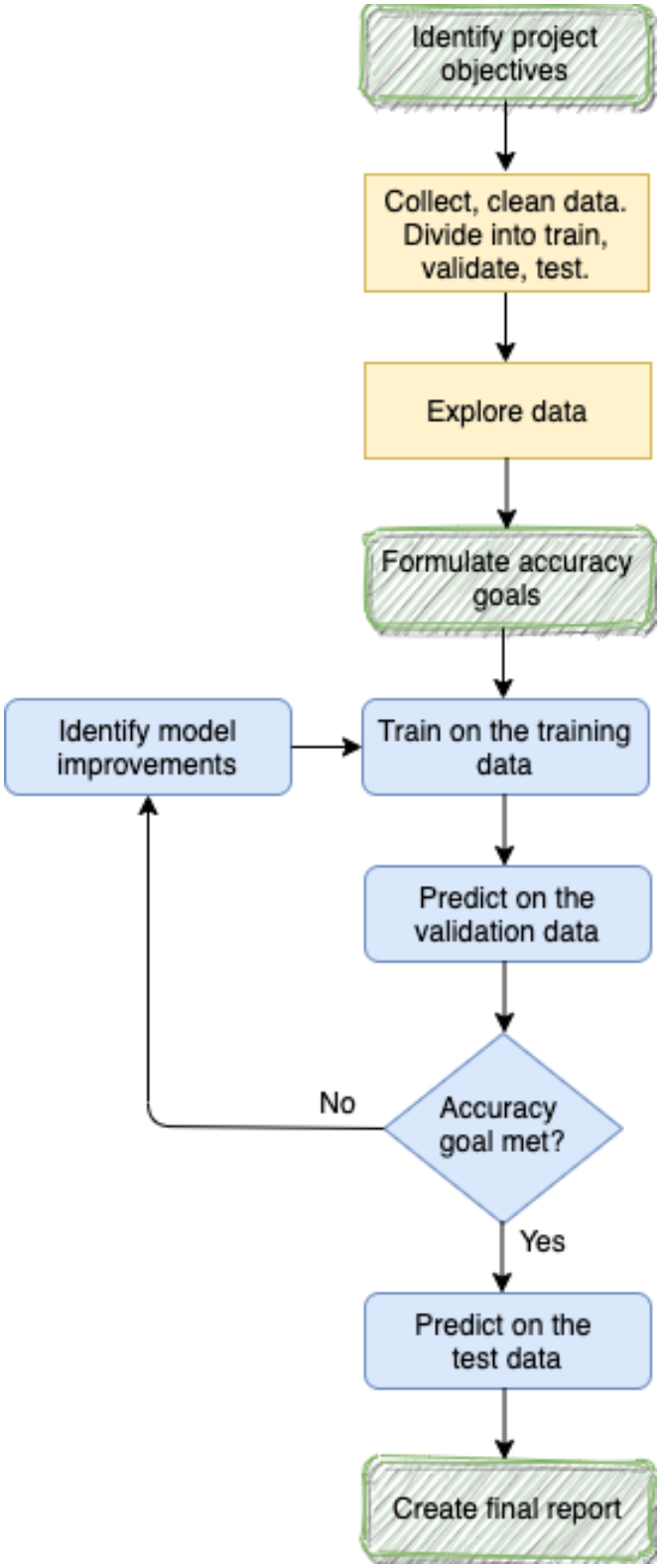


Figure 5.1: Simplified Workflow for a Machine Learning Project

Part II: Linear Models

Preface to Part Two

Part Two explores *linear model* machine learning algorithms, including:
- linear regression
- logistic regression
- naive Bayes

These algorithms are grouped together in Part Two because they take a linear combination of inputs to estimate either a target output variable or a linear decision boundary for a given input data set $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$:

$$y = \mathbf{w}\mathbf{X} + b$$

where $\mathbf{w}$ represents the weight matrix which is multiplied by the $\mathbf{X}$ input features plus the fitting parameter b. The parameters w and b are learned from the data D.

Keep in mind that *linear* does not always mean a straight line, but any line that can be created with a linear combination of inputs. For example, $y = x^2$ is a line, but not a straight line.

In linear regression, the target output y will be a real-number value. We call this kind of machine learning *regression*. In logistic regression, the target output y will be an indicator for membership in one of a finite number of classes. We call this kind of machine learning *classification*. In logistic regression and naive Bayes, the output is the *probability* of membership in one of a finite number of classes.

These models can be considered *parametric* algorithms because we have a fixed set of parameters, or weights, learned from the data. In contrast, with non-parameteric methods in which the number of parameters grows with an increase in data size.

We begin with linear models because they have visual, intuitive explanations and are commonly used in machine learning tasks.

6. Linear Regression

6.1 Overview

Many machine learning algorithms have their origin in the field of statistics, and linear regression is a prime example. In fact, the fundamentals of linear regression can be traced back to mathematicians in the early 1800s. We begin with linear regression because it is relatively simple, yet powerful, and introduces foundational concepts and terminology we will use for other algorithms throughout the book. In linear regression, our data consists of predictor values, x, and target values y. We wish to find the relationship between x and y. This linear relationship can be defined by parameters w and b, with w, the slope of the line, quantifying the amount that y changes with changes in x, and b serving as an intercept.

6.2 Linear Regression in R

Let's take a look at the `women` data set, one of the built-in data sets in R.

Code 6.2.1 — women data. Height in inches and weight in pounds for 15 women.

```
# explore at the console:
> str(women)
'data.frame': 15 obs. of  2 variables:
 $ height: num  58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
> plot(women$weight~women$height)
```

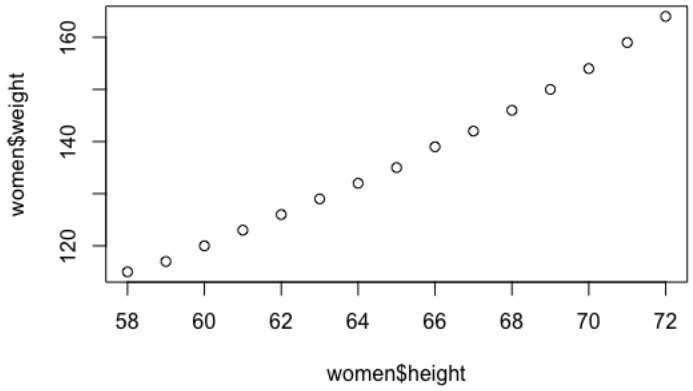


Figure 6.1: Women Data Set

This data set is from the mid-1970s, when Americans were considerably thinner than we are now. Figure 6.1 reveals a linear trend in the data: taller women weigh more than shorter women.

We will use the built-in lm() function to build a linear regression model. There are many parameters we can send to the lm() function, here we only send two: the formula, and the data. The formula *weight* ~ *height* says to model weight, our target, as a function of height.

Code 6.2.2 — Build a linear regression model. You can type `?lm` at the console to learn more about the function.

```
> lm1 <- lm(weight~height, data=women)
> lm1

Call:
lm(formula = weight ~ height, data = women)

Coefficients:
(Intercept)       height
     -87.52         3.45
```

When we built the model, we saved it to variable `lm1`. If we type the model name at the console as we see in the code above, we get basic information about the model. Our parameters (coefficients) are the w and b that the algorithm learned from the data. In this model, they are $w = 3.45$, and $b = -87.52$. The intercept parameter, b, is generally not of interest to us as it is just used to fit the data. The parameter w is of more interest. In linear regression, it tells us how much we can expect the y-value to change for every one-unit change in the x-value. So for every inch taller a woman is, we expect her to weight 3.45 pounds more. We can use these values for prediction. Let's say we have a woman who is 65 inches tall. Her weight should be:

$65 * 3.45 - 87.52 = 136.7$. Check if that value makes sense given the data in Figure 6.1.

We have a model but we don't know if it is a good model. We can use the summary() function to get a glimpse of the goodness of fit achieved by this model. This is shown in the next code block. We will delve deeper into the summary() output later but now we will just point out two key elements of the output. Notice the three asterisks at the end of the line for height under Coefficients. This indicates that height was a good predictor. Notice also that the intercept received three asterisks, but we are not really interested in the intercept, just the relationship between height and weight. The second thing to notice is the R-squared is about 0.99. The R-squared is a measure of goodness of fit that ranges from 0 to 1, the closer to 1 the better. This provides evidence that we have a good model for this data.

Code 6.2.3 — Model summary. Use the summary() function to learn more about the model.

```
> summary(lm1)
Call:
lm(formula = weight ~ height, data = women)

Residuals:
    Min      1Q  Median      3Q     Max
-1.7333 -1.1333 -0.3833  0.7417  3.1167

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.51667    5.93694  -14.74 1.71e-09 ***
height        3.45000    0.09114   37.85 1.09e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991,Adjusted R-squared:  0.9903
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

Exercise 6.1 — Building a Simple Linear Regression Model. Using the built-in data set `swiss`. Try the following in an R script:

- Learn more about the data by typing ?swiss at the console.
- Load the data and look at it with R data exploration functions and at least one plot.
- Build a linear regression model predicting Fertility based on Education.
- What is the mean of Education? of Fertility?
- What is the predicted Fertility, given the mean of Education? Use the coefficients from your model.
- Do you think Education is a good predictor? Why or why not?
- Do you think this is a good model? Why or why not?

6.3 Metrics

There are a lot of metrics associated with linear regression and we will start with metrics we can use in data analysis before we begin applying a machine learning algorithm.

6.3.1 Metrics for Data Analysis

In linear regression we often want to know if variables are correlated. We can use the cor() function or plot pairs() as shown in Section 2.4 earlier. For example, to see if x and y are correlated we can use this code: `cor(x, y)`.

The default method is Pearson's, which measures the linear correlation between two variables. It ranges from -1 to $+1$ where the former is a perfect negative correlation, the latter is a perfect positive correlation, and values close to 0 indicate little correlation. The formula for Pearson's correlation is:

$$\rho_{x,y} = Corr(x,y) = \frac{Cov(x,y)}{\sigma_x \sigma_y} \tag{6.1}$$

We see that correlation is actually covariance, scaled to $[-1,1]$. Covariance measures how changes in one variable are associated with changes in a second variable. The numbers can range wildly which is why the scaled correlation is often preferred. Here is the formula for covariance, where n is the number of data points:

$$cov(x,y) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n-1} \tag{6.2}$$

6.3.2 Linear Regression Metrics for Model Fit

The summary() output of a linear model gives a lot of useful metrics concerning the model fit. Looking back at the output of summary() for the linear model at the Coefficients section:

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.51667    5.93694  -14.74 1.71e-09 ***
height        3.45000    0.09114   37.85 1.09e-14 ***
```

The estimated coefficient for height and the intercept are given, along with standard error, t value and p value. The standard error gives us an estimate of variation in the coefficient estimate and can be used to predict a confidence interval for the coefficient. So the confidence interval for w_1 would be: $w_1 \pm 2\, SE(w_1)$. Standard errors are used for the hypothesis test on the coefficient, where the null hypothesis is that there is no relationship between the predictor variable and the target variable. In other words, the true $w = 0$. This is computed using the t-statistic:

$$t = \frac{\hat{w}_1 - 0}{SE(\hat{w}_1)} \tag{6.3}$$

which measures the number of standard deviations our estimate coefficient $\hat{w}_1$ is from 0. Notice we put the hat symbol, ˆ, over w to remind us that it is an estimate. The distribution of the t-statistic has a bell shape which makes it easy to compute the probability of observing a t-statistic larger in absolute value than what was computed, if the null hypothesis were true. This is the p-value. If the p-value is small we can reject the null hypothesis. Typical cut-off points for the p-value are 0.05 and 0.01. One caveat about p-values is that generally you will have more confidence in them if your data size is greater than 30. That is not the case for the women data because it has only 15 observations.

The final part of the summary() output of the women linear regression model is:

```
Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991,Adjusted R-squared:  0.9903
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

Whereas the metrics for the coefficients indicate how well each coefficient modeled the true data, these statistics tell us how well the model as a whole fit the training data. The RSE, residual standard error, is computed from the RSS, residual sum of squares.

$$RSS = e_1^2 + e_2^2 + ... + e_n^2 = \sum_i (y_i - \hat{y}_i)^2 \tag{6.4}$$

The RSS is just the sum of squared errors. We square them because some will be errors in the positive direction and some will be in the negative direction. The RSE is computed from the RSS:

$$RSE = \sqrt{\frac{1}{n-2} RSS} \quad = \quad \sqrt{\frac{1}{n-2} \sum_i (y_i - \hat{y}_i)^2} \tag{6.5}$$

Why is RSS scaled by $\frac{1}{n-2}$? The value n is the number of observations and the 2 is because we have 2 estimated variables. This gives us $15 - 2 = 13$ degrees of freedom mentioned on the line with the RSE. The RSE measures how off our model was from the data, the lack of fit of the model. It is measured in units of y so in this case our RSE of 1.525 is about 1.5 pounds.

Since RSE is in terms of Y it can be hard to interpret when we have multiple predictors. For this reason the R^2 statistic is also provided:

$$R^2 = 1 - \frac{RSS}{TSS} \tag{6.6}$$

where TSS, total sum of squares, is a measure of how far off y values tend to be from the mean:

$$TSS = \sum (y_i - \bar{y})^2 \tag{6.7}$$

The R^2 statistic will always be between 0 and 1, the closer to 1 the more variance in the model is explained by the predictors. The R^2 in the summary above was 0.991, which is very high. This means that almost all the variation in weight is predicted by height. For linear regression models such as this one where there is only one predictor, R^2 will be the same as the squared correlation between the X and Y values.

The final statistic listed is the F-statistic:

$$F = \frac{(TSS - RSS)/p}{RSS/(n-p-1)} \tag{6.8}$$

where n is the number of observations and p is the number of predictors. The F statistic takes into account all of the predictors to determine if they are significant predictors of Y. It provides evidence against the null hypothesis that the predictors are not really predictors. The advantage of the F-statistic over R^2 is that R^2 does not tell us whether it is statistically significant or not but the F-statistic does. It has an associated p-value. So we check for a F-statistic greater than 1 and a low p-value to indicate confidence in the model.

6.3.3 Metrics for Test Set Evaluation

Earlier we ran the women data set through the linear regression algorithm and looked at some metrics for judging the quality of the model. In practice, we will not run entire data sets through the algorithm, but divide the data into a training set and a test set. The model should be built on the training data and should not see the test data until evaluation time. Then the model is used to predict values for the test data, and we can use various metrics to see how far off the predictions were from the true values. We will do that in future examples. For now, we are going to just make up some test data out of the blue, making sure that the data does not fit well to the regression line so it shows up well on the graph in Figure 6.2. Again, we are only making up data for illustration purposes.

What are some metrics we can use to evaluate the prediction accuracy? For regression tasks, common metrics include mean squared error and correlation. Imagine that we randomly selected a few data observations to hold out from the training data. We call this held-out data a *test set*, or *validation set*. These observations would not have been seen by the algorithm during training and so we can use them to test the model. It is very important that the algorithm is tested on unseen data, otherwise we don't know if the algorithm learned anything or just memorized data. Using the test data to predict weight, given height, we can compute the correlation of the actual y values with the predicted y values via the R cor() function: $cor(predicted, actual)$. Correlation gives us a general idea about our model. A correlation close to +1 would mean that as height went up, weight went up as well. However, this would not tell us how much our estimates were off for the individual observations. These errors are called residuals:

```
residuals <- predicted - actual
```

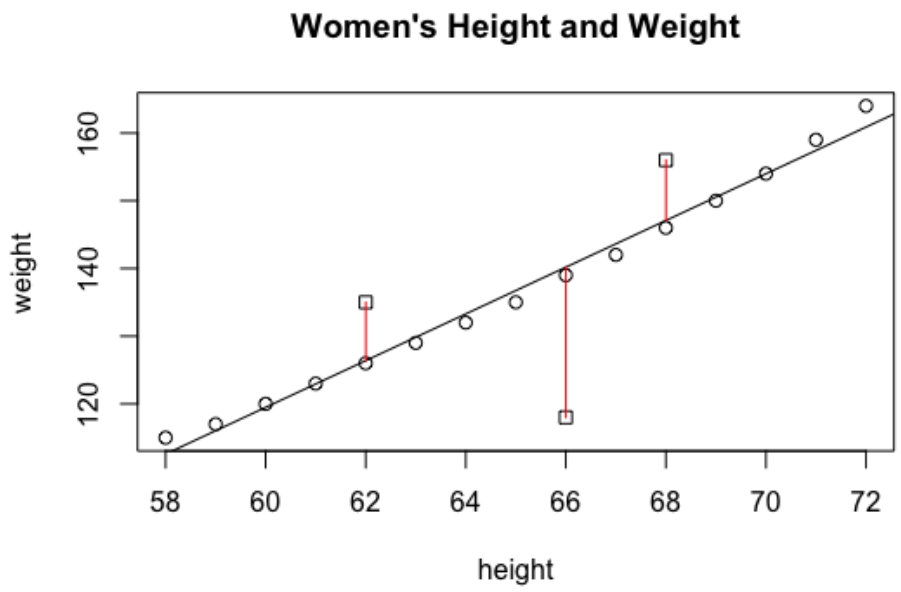


Figure 6.2: Test Set Errors

In Figure 6.2 these residuals are represented as vertical lines drawn from the data points to the regression line. Some errors may be in the positive direction and some may be in the negative direction. For this reason they are squared to find mean squared error:

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6.9}$$

This averages the squared difference between the actual values (y) and the predicted values (y-hat) over all elements in the test set. Sometimes the square root of the above is used, rmse (root mean squared error), since it will be in units of y. The metrics mse or rmse are useful in comparing two models built on the same training data.

We see in Code 6.3.1 that our correlation is 0.38 which is not good, unsurprisingly, since we purposely made up data that was far from the regression line. As you can see in Figure 6.2, the 3 test examples, shown as squares, are not that close to the regression line. Two are above the line and one is below. Red vertical lines show the residuals, the errors. We quantify the amount of error with mse and rmse. The mse value can be hard to interpret in isolation; it is most useful in comparing models. The rmse however is in units of y. In this case, we see that our test data was off by 14.67 pounds on average.

Again, it should be stressed that we normally don't hallucinate test data. We normally take our data and divide it into train and test sets.

Code 6.3.1 — Hallucinated Test Data. Women Height-Weight Model.

```
# fake test data
test <- women[c(5, 9, 11),]
test[1, 2] <- 135
test[2, 2] <- 118
test[3, 2] <- 156
# predictions on test data
pred <- predict(lm1, newdata=test)
# metrics
correlation <- cor(pred, test$weight)
print(paste("correlation: ", correlation))
mse <- mean((pred - test$weight)^2)
print(paste("mse: ", mse))
rmse <- sqrt(mse)
print(paste("rmse: ", rmse))
[1] "correlation:  0.38404402702441"
[1] "mse:  215.284722222223"
[1] "rmse:  14.6725840335717"
```

Exercise 6.2 — Exploring Metrics. Using the linear regression model you built on the swiss data, try the following. For many of the following steps you will need to refer back to the formulas in this section.

- Run the R covariance and correlation functions on Education and Fertility and discuss your observations.
- Does the order of the operands matter for these functions?
- Create R code to compute the t-value for the estimated Education coefficient, and compare to the summary output for the model.
- Create R code to compute rss, rse, and r-squared for the model, and compare to the summary output for the model.
- Make a test set of 5 randomly chosen observations from the swiss data set. Note: This is cheating! We never let the algorithm see the test data before evaluation. However, we are just using the "test" set to compute some more metrics.
- Make predictions on Fertility for this "test" data.
- Print the test and predicted values side-by-side using print() and cbind(). Are they as close in value as you would expect?
- Run cor() and compute mse on the predicted versus actual data.
- Compute the rmse and compare this to the mean of the absolute value of the residuals. Are they close? Is this surprising? Why or why not?

▪

6.4 The Algorithm

The example above is called *simple linear regression* because it had only one predictor variable, height, for the target variable, weight. When we have more than one predictor variable as we will see below, it is called *multiple linear regression*. In general, the more predictors are added to a model, the better it will be, but that does not mean you should just add all the predictors. We will learn techniques for determining which predictors should be included as we go.

Recall that the coefficients for the simple linear regression model above were $w = 3.45$ and $b = -87.51667$. Therefore, looking at an observation where height is 64 inches, we would expect weight to be:

$3.45 * 64 - 87.511667 = 133.28$

and we see from Figure 6.2 that this is a reasonable estimate of weight for this height. These parameters w and b were learned from the training data, but how? We will dig deeper into the math below, but the general idea is that we want a line than minimizes the residuals, the errors, designated as e:

$$\underset{w,b}{\operatorname{argmin}} \sum_{i=1}^{n} e_1^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{6.10}$$

The above equation states that we want the parameters w, b that minimize the squared errors over all n examples in the training data. The estimated values of w and b are:

$$\hat{b} = \bar{y} - \hat{w}\bar{x} \qquad \hat{w} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{6.11}$$

As seen above, these equations rely on mean values of x and y to compute w and b. To prove to ourselves that these equations find the parameters w, b, let's find the mean of x and y in the women's data and plug them into the equations.

Code 6.4.1 — Verifying the equations. Manually Computing Coefficients.

```
x <- women$height
y <- women$weight
x_mean <- mean(women$height)
y_mean <- mean(women$weight)

w_hat <- sum((x-x_mean)*(y-y_mean)) / sum((x-x_mean)^2)
b_hat <- y_mean - w_hat * x_mean
print(paste("w and b estimates = ", w_hat, b_hat))

[1] "w and b estimates =  3.45 -87.5166666666667"
```

Exercise 6.3 — Verify the equations. Using the formulas in this section in R, verify the coefficients for the linear model you built on the swiss data. Comment on the role of means in linear regression. ▪

The algorithm described above for linear regression is called the **ordinary least squares (OLS) method**. Next we explore how these equations are derived.

6.5 Mathematical Foundations

Our goal in the OLS method is to reduce the errors over all the training data. These errors are quantified in the residual sum of errors, RSS:

$$RSS = e_1^2 + e_2^2 + ... + e_n^2 \tag{6.12}$$

Each error, e, in turn is the difference between the actual y value and the predicted value:

$$RSS = (y_1 - b - wx_1)^2 + (y_2 - b - wx_2)^2 + ... + (y_n - b - wx_n)^2 \tag{6.13}$$

The **loss function** describes how much accuracy we lose in our model. The first equation below specifies the loss for one example, the second averages the loss over all the examples. By the way, you will also see this called a **cost function**. The terms loss function, cost function, and error function are used somewhat synonymously, but unfortunately inconsistently across the literature. In the equations below we see the loss function with subscript i to indicate the loss for one instance, and the loss function without the subscript indicates the loss averaged over all examples.

$$\mathcal{L}_i = (y_i - f(x_i))^2 \tag{6.14}$$

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{6.15}$$

Our goal is to find the parameters (coefficients) that minimize these errors on the training data:

$$\underset{w,b}{\text{argmin}} \sum_{i=1}^{n} e_1^2 = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6.16}$$

One method for finding the coefficients is to take the partial derivatives of the loss function, set them to zero, and solve. This produces the normal equations given in the last section. Here

we show the details of how they are derived. The equations above express the loss function in algrebraic notation. We could find the derivative with this notation but it is more concise if we use matrix notation. We are now going to consider parameter b to be one of possibly many parameters (in this case only two) in a vector. Specifically, we will refer to parameter b as w_0. Additionally we will express x as a vector, making the first element 1 so that it multiplies by w_0, the intercept.

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \end{bmatrix}$$

$f(x) = \mathbf{w}^T\mathbf{x} = w_0 + w_1 x_1$

In the notation above, only one predictor is used. For multiple predictors, the w and x vectors would expand accordingly. Let's visualize this matrix representation of f(x) for the women data set.

$$\mathbf{y}\begin{bmatrix} 115 \\ 117 \\ 120 \\ 123 \\ 126 \\ ... \\ 154 \\ 159 \\ 164 \end{bmatrix} = \quad \mathbf{w}\begin{bmatrix} -87.5167 \\ 3.45 \end{bmatrix}^T \quad \mathbf{X}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & ... & 1 & 1 & 1 \\ 58 & 59 & 60 & 61 & 62 & ... & 70 & 71 & 72 \end{bmatrix}$$

Our loss function expressed in matrix notation, with i indexing indivdual observations:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{n}(\mathbf{y}_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{6.17}$$

Rewriting the square term above into the form shown below uses the property that $(\mathbf{Xw})^T = \mathbf{w}^T\mathbf{X}^T$:

$$\mathcal{L} = \frac{1}{N}(\mathbf{y} - \mathbf{Xw})^T(\mathbf{y} - \mathbf{Xw}) \tag{6.18}$$

Next we go through a few steps to multiply out and collect terms. First, bring the transpose inside the parenthesis.

$$\mathcal{L} = \frac{1}{N}(\mathbf{y}^T - (\mathbf{Xw})^T)(\mathbf{y} - \mathbf{Xw}) \tag{6.19}$$

Multiply out:

$$\mathcal{L} = \frac{1}{N}\mathbf{y}^T\mathbf{y} - \frac{1}{N}\mathbf{X}\mathbf{w}\mathbf{y}^T - \frac{1}{N}(\mathbf{X}\mathbf{w})^T\mathbf{y} + \frac{1}{N}(\mathbf{X}\mathbf{w})^T\mathbf{X}\mathbf{w} \tag{6.20}$$

We can combine the middle two terms because $w^T X^T y$ and $y^T Xw$ are transposes of one another and scalars.

$$\mathcal{L} = \frac{1}{N}\mathbf{y}^T\mathbf{y} - \frac{2}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \frac{1}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} \tag{6.21}$$

Before we take the partial derivative wrt w, we can get rid of the first term since it doesn't involve w.

$$\mathcal{L} = -\frac{2}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \frac{1}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} \tag{6.22}$$

Now use the rules for matrix partial derivatives to get:

$$\frac{\partial\mathcal{L}}{\partial w} = -\frac{2}{N}\mathbf{X}^T\mathbf{y} + \frac{2}{N}\mathbf{X}^T\mathbf{X}\mathbf{w} \tag{6.23}$$

Above, we used the fact that the partial derivative of $w^T x = x$ for the first term and $w^T w = 2w$ for the second term. We simplify the equation to:

$$\mathbf{X}^T\mathbf{y} = \mathbf{X}^T\mathbf{X}\mathbf{w} \tag{6.24}$$

And so our estimated parameters must be:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{6.25}$$

In practice, finding the inverse matrix above is computationally intense, $O(n^3)$, and therefore very slow for large data sets. For this reason, R uses optimization techniques to find the parameters.

6.5.1 Gradient descent

The situation just described in which a direct mathematical approach would be computationally expensive is a common one. This has led to the development of many optimization techniques. In this section we discuss gradient descent, and as we will see through the book, it is one of the most commonly used optimization techniques. Unlike directly finding the inverse in the normal equation above, gradient descent will not get bogged down for large data sets. The algorithm starts with some value for the parameters $\mathbf{w}$ and keeps changing them in an iterative

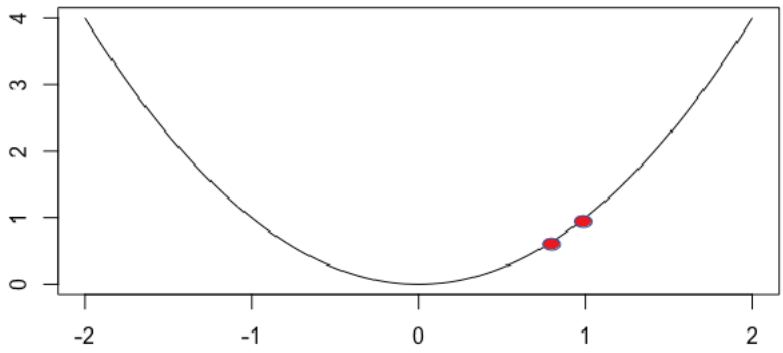


Figure 6.3: Searching for Error Minimum

loop until they find a minimum. Figure 6.3 visualizes a convex function with a random starting point symbolized by the higher red dot. One iteration may move the first dot to the second location. This is one step.

The gradient descent algorithm repeats the parameter update until convergence:

$$w_j := w_j - \alpha \frac{\partial \mathcal{L}}{\partial w} \tag{6.26}$$

Note that all parameters in w are updated at every iteration. The derivative gives us the slope and the alpha determines our step size. If alpha is too small, the algorithm will be slow. On the other hand, if it is too large we could overshoot the minimum and fail to converge.

Exercise 6.4 — Mathematical Foundations. Use what you have learned in this section to answer the following questions. A good reference for R matrix operations is: `https://www.statmethods.net/advstats/matrix.html`

- What is the purpose of a loss or cost function?
- Is gradient descent guaranteed to find the optimal parameter? Why or why not?
- Using R matrix operations, compute the w matrix of coefficients using formula 5.25 above.

6.6 Multiple Linear Regression

Next we look at another example of linear regression in R. We will use the R build-in data set ChickWeight, which has 578 rows and 4 columns of data resulting from an experiment on the effect of different types of feed on chick weight. We will use weight as our target, with the following predictors:

- Time - number of days since birth
- Diet - a factor representing 4 different diets

We will ignore the Chick column which identifies the chicken. In **simple linear regression** we use only one predictor. In **multiple linear regression** we use more than one predictor. We will do both on this data set. First, let's make a couple of plots to visualize the data.

Code 6.6.1 — Plots. Use par() to set up a 1x2 grid for the plots.

```
par(mfrow=c(1,2))
plot(ChickWeight$Time, ChickWeight$weight,
     xlab="Time", ylab="Weight")
plot(ChickWeight$Diet, ChickWeight$weight,
     xlab="Diet", ylab="Weight")
```

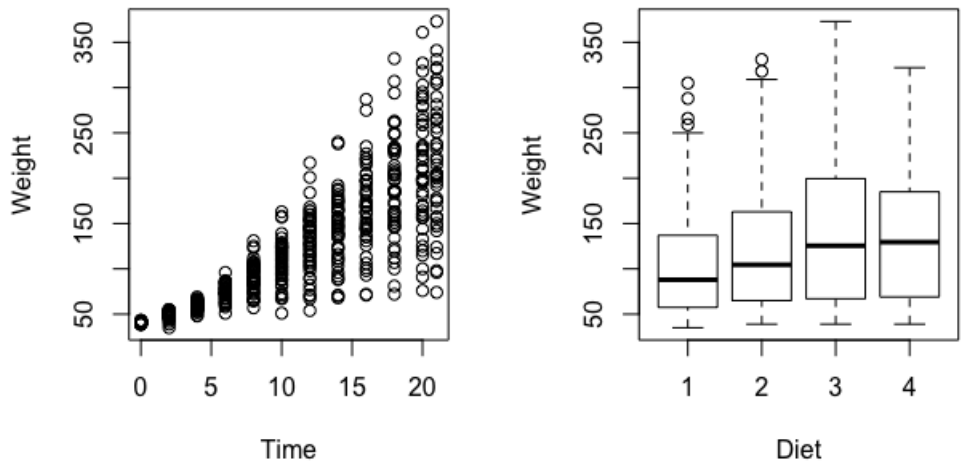


Figure 6.4: ChickWeight Weight Impacted by Time, Diet

In Figure 6.4 we see in the plot on the left that chicks gain wait over time. That is not surprising but what the plot also shows is that chicks start off at near identical weights and diverge over time. The plot on the right shows the impact of diet on weights. Box plots show the median value as the horizontal line through the box, the box itself shows the IQR (inter-quartile range from the first to the third quartile), and the horizontal lines at the end of the dashed lines show minimum and maximum values, not including suspected outliers which appear as dots beyond the horizontal lines. In this example it appears that diet 4 is slightly better than diet 3 and diet 1 appears to result in the lowest weights for chicks.

Next we divide the data into train and test sets by randomly sampling the rows. We set a seed so that each time we run this code we should get the same results. Vector i will contain the row numbers so we can subset the data frame with indices i to get a train set and *not i, or -i* to get the test set. After this 75/25 split, the train set has 433 observations and the test set has 145.

Code 6.6.2 — Divide data into train and test sets. Made a 75/25 split.

```
set.seed(1234)
i <- sample(1:nrow(ChickWeight), nrow(ChickWeight)*0.75,
     replace=FALSE)
train <- ChickWeight[i,]
test <- ChickWeight[-i,]
```

The sample() function as we used it above has this form: `sample(x, size, replace = FALSE)`. The first argument is a vector of elements from which to choose, in our case from 1:578, the row numbers of the data frame. The second argument specifies the size; in our case we want 75% of the row numbers. The last argument indicates that sampling should be done without replacement. To learn more about this function type `?sample()` at the console.

6.6.1 Interpreting summary() output for a linear model

Next we create a linear model on the train set, using only Time as a predictor. Let's look at the output of summary() in more detail. First it echoes back the code used to build lm1.

Next we see the distribution of the residuals, the errors. We want to see that the residuals are symmetrically distributed around the mean or median. There is a wide range here, from the minimum of -140 to the maximum of 158. That is not encouraging. The wide range of residuals confirms what we saw in Figure 6.4, that chick weight diverges widely as time goes on.

The Coefficients section lists for each predictor and the intercept, the estimated value, standard error, t value and p value, followed by significance codes. If you recall from our earlier discussion, a t-value measures variation in the data, and the associated p-value estimates confidence in that value. A low p-value indicates evidence for rejecting the null hypothesis that the predictor does not influence the target variable. We want to see low p-values and in this case we do. Time has a low p-value and correspondingly, 3 asterisks. The estimate for our one predictor, Time, is 8.952. This means we would expect chicks to gain an average of almost 9 gm a day. The standard error measures the average amount the coefficient estimate varies from the actual values. The t-value is a measure of how many standard deviations the coefficient estimate was from 0. The further it is from 0, the more confidence we have in rejecting the null hypothesis, in this case that Time has no effect on chick weight. The p-value gives the probability of observing a similar or larger t-value due to chance, given the data. A small p-value gives us confidence that there really is a relationship between our predictor(s) and the target variable. The chart for the significance codes is given at the bottom of the Coefficients section.

The last section gives some statistics on the model. The residual standard error, RSE, is in units of y. In this case our RSE was 41.4, so the average error of the model was about 41 gm. This statistic was calculated on 431 degrees of freedom: we had 433 data points minus 2 predictors. Multiple R-squared is scaled from 0 to 1 and so is easier to interpret than RSE. The adjusted R-squared is 0.6863 which is not bad. This means that 68% of the variance in the model can be explain by our predictor. The adjusted R-squared takes into account the number

of predictors. This is important because R-squared tends to increase as we add predictors, and the adjusted R-squared accounts for this. Finally the F-statistic also measures whether our predictors and target are related. We want to see the F-statistic be far away from zero and its associated p-value to be low. The more data observations we have, the lower the F-statistic can be to confirm the relationship. This is why the p-value is important.

Code 6.6.3 — Linear model 1. Using only Time as a predictor.

```
lm1 <- lm(weight~Time, data=train)
summary(lm1)
```

```
Call:
lm(formula = weight ~ Time, data = train)
Residuals:
     Min       1Q   Median       3Q      Max
-140.314  -16.648    0.778   14.682  158.686
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   26.318      3.743   7.031 8.06e-12 ***
Time           8.952      0.291  30.760  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
Residual standard error: 41.4 on 431 degrees of freedom
Multiple R-squared:  0.687,Adjusted R-squared:  0.6863
F-statistic: 946.2 on 1 and 431 DF,  p-value: < 2.2e-16
```

6.6.2 Residuals

Plotting the residuals results in 4 plots, which we have arranged in a 2x2 grid in Figure 6.5. How do we interpret these plots? A comprehensive explanation is given here: `http://data.library.virginia.edu/diagnostic-plots/`.

Code 6.6.4 — Plot the residuals. The residuals give us information about how well the model fits the data.

```
par(mfrow=c(2,2))
plot(lm1)
```

Below we provide a brief overview of the 4 plots:

1. Plot 1 Residuals vs Fitted: This plots the residuals (errors) with a red trend line. You want to see a fairly horizontal red line. Otherwise, the plot is showing you some variation in the data that your model did not capture.
2. Plot 2 Normal Q-Q: If the residuals are normally distributed, you will see a fairly straight diagonal line following the dashed line.
3. Plot 3 Scale-Location: You want to see a fairly horizontal line with points distributed equally around it. If not, your data may not be homoscedastic (means "same variance").

4. Plot 4 Residuals vs Leverage: This plot will indicate leverage points which are influencing the regression line. They may or may not be outliers, but further investigation is warranted. An **outlier** is a data point with an unusual y value whereas a **leverage point** is a data point with an unusual x value.

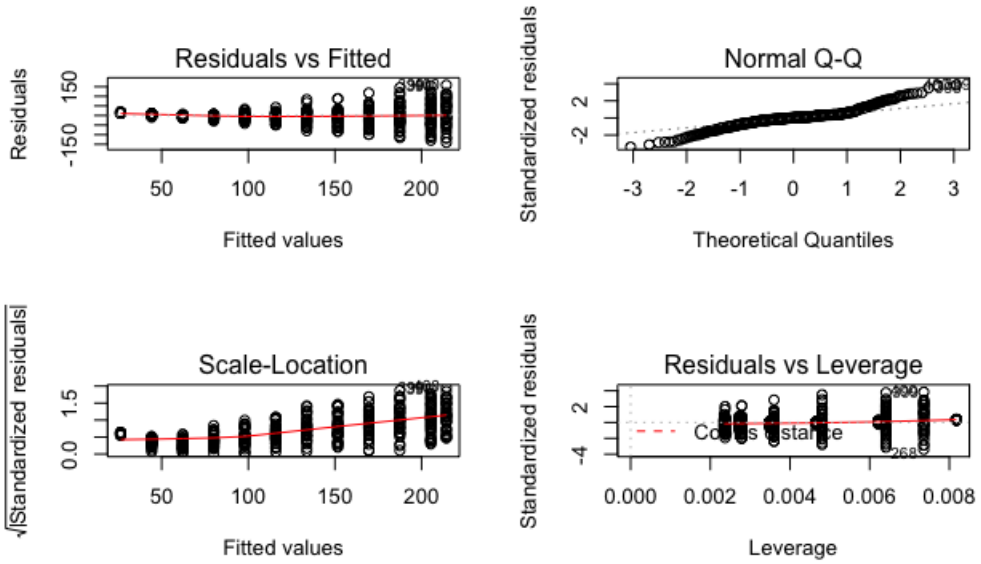


Figure 6.5: Residuals Plot

Looking at Figure 6.5 we see some problems with our model. In the first graph, the red line is horizontal but notice that the residuals vary more as we go to the right. This confirms our very first observation of the data, that chicks vary greatly in how much weight they gain. Time and even diet cannot account for this. What is not accounted for in our model is the genetic contribution. We humans start off life at an average of about 7 pounds but end up as adults in a wide range of weights. We could consider genetics a hidden or unseen variable in this experiment. Also chick gender was not included in the data, so we have no way of knowing if this influenced the weight variation. The second graph indicates that most of the residuals are normally distributed except for those in the lower range. So there is variation in the data that our model does not capture.

Let's build another model using Time and Diet as predictors. We do this with the plus sign in the formula: Time+Diet. The adjusted R-squared for lm2 is 0.7338 which is higher than the adjusted R-squared for lm1, which was 0.6863. Also, RSE has decreased to 38.13 from 41.1 in lm1. Adding Diet seems to have improved our model.

Code 6.6.5 — Linear model 2. Using Time and Diet as predictors.

```
lm2 <- lm(weight~Time+Diet, data=train)
summary(lm2)
```

```
Call:
lm(formula = weight ~ Time + Diet, data = train)

Residuals:
     Min       1Q   Median       3Q      Max
-137.857  -20.492   -1.685   16.955  137.365

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   8.4109     4.1372   2.033 0.042670 *
Time          8.9086     0.2682  33.218  < 2e-16 ***
Diet2        16.3645     4.9235   3.324 0.000965 ***
Diet3        40.1424     4.8907   8.208 2.67e-15 ***
Diet4        32.1873     5.2503   6.131 1.99e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 38.13 on 428 degrees of freedom
Multiple R-squared:  0.7363,Adjusted R-squared:  0.7338
F-statistic: 298.8 on 4 and 428 DF,  p-value: < 2.2e-16
```

6.6.3 Dummy variables

Looking at the summary output for lm2, we see that Time and Diet were good predictors. Why do we have Diet2, Diet3, Diet4? Since Diet is a factor with 4 levels, R made 3 **dummy variables** for us. The base model represents Diet1, the dummy variables Diet2 - Diet 4 tell us how much each diet impacted the model compared to Diet 1. Recall from the boxplot above, that Diet1 resulted in the lowest weights and we see that confirmed in this summary, because Diet2 - Diet4 each have positive coefficients. For each data observation, only one of the dummy variables will be active with the others being zero. So for a chick on Diet 1, the dummy variables for Diets 2 through 4 would be zero.

6.6.4 The anova0 function

We can compare the summary() statistics of models to gauge their relative value. Another way to compare them is to run anova() on the two models. The anova() function lists each model and provides similar statistics as the summary() function for each model. We see that the RSS is lower for model 2, and model 2 is given a low p-value. This is confirmation that lm2 outperformed lm1.

Code 6.6.6 — The anova() function. Analysis of Variance.

```
anova(lm1, lm2)
```

```
Analysis of Variance Table
```

```
Model 1: weight ~ Time
Model 2: weight ~ Time + Diet
  Res.Df    RSS Df Sum of Sq      F    Pr(>F)
1    431 738546
2    428 622323  3    116222 26.644 8.107e-16 ***
```

Let's try one more thing. Recall from Figure 6.5 that there was a funnel shape in the residuals in that they became more spread out from left to right. The log function damps down x values across the axis. Perhaps this could squish the chick weights closer about the linear regression line.

Code 6.6.7 — Linear model 3. Linear models are not always a straight line.

```
lm3 <- lm(log(weight)~Time+Diet, data=ChickWeight)
summary(lm3)
```

```
Residual standard error: 0.2281 on 573 degrees of freedom
Multiple R-squared:  0.8484,Adjusted R-squared:  0.8474
F-statistic:   802 on 4 and 573 DF,  p-value: < 2.2e-16
```

Above we show only the statistics portion of the summary() output. Now the R-squared has increased to 0.8474, indicating that lm3 may be better than lm1 and lm2. We cannot run anova on the 3 models because lm3 has a different response: log(weight) instead of weight. But we could look at the residuals plots to look for improvement.

Exercise 6.5 — Multiple Linear Regression. Using the same swiss data, try the following activities:

- Run cor() and pairs() on swiss and discuss any patterns you see.
- Build a linear regression model predicting Fertility from all predictors.
- Build another model just using 2 or 3 predictors of your choice.
- Compare RSE and R-squared for the simple linear regression model and the two models you just created. Which appears best?
- Run anova() on the 3 models and discuss the results.
- Plot the residuals on your best model and discuss what you see.

6.7 Polynomial Linear Regression

To emphasize the point that linear regression is not always a straight line, we next look at polynomial linear regression. We will perform polynomial regression on the cars dataset, included in R. The data set has 50 observations and 2 variables: speed and stopping distance. The code example is from the R documentation. The plot() call at the top of the code sets up the plot. The seq() call sets up a sequence for the s values we want to plot across the horizontal axis. The for loop plots models of degree 1 through 4 in different colors. Colors 1-4 correspond to black, red, green3, blue. The poly() function is used to create orthogonal (not correlated) polynomials. Finally an anova() is run on the 4 models.

Code 6.7.1 — Polynomial Regression. Using the cars data set

```
plot(cars, xlab = "Speed (mph)",
    ylab = "Stopping distance (ft)", xlim = c(0, 25))
s <- seq(0, 25, length.out = 200)
for(degree in 1:4) {
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep = "."), fm)
  lines(s, predict(fm, data.frame(speed = s)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)
```

```
Analysis of Variance Table

Model 1: dist ~ poly(speed, degree)
Model 2: dist ~ poly(speed, degree)
Model 3: dist ~ poly(speed, degree)
Model 4: dist ~ poly(speed, degree)
  Res.Df   RSS Df Sum of Sq      F Pr(>F)
1     48 11354
2     47 10825  1    528.81 2.3108 0.1355
3     46 10634  1    190.35 0.8318 0.3666
4     45 10298  1    336.55 1.4707 0.2316
```

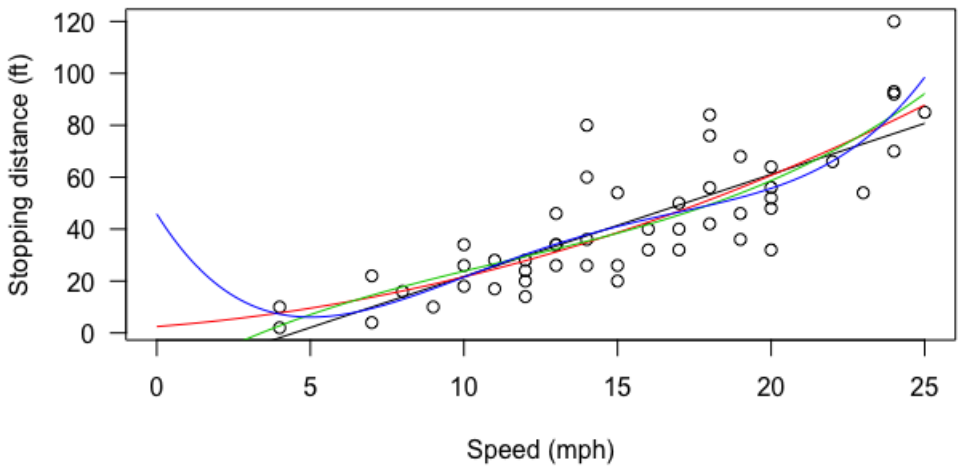


Figure 6.6: Cars with Polynomial Degree 1:4

When we look at the results above,[1] we see that the residuals of Model 2 are less than the degree-1 model. This is an indication that model 2 might be better than model 1. Residuals for models 3 and 4 are lower still, but at some point we worry about overfitting the training data, which we will discuss in the next section.

Exercise 6.6 — Polynomial Linear Regression. Using the same swiss data, try the following activities:

- Rerun the degree-1 linear regression model predicting Fertility from Education if necessary.
- Build a degree-2 regression model predicting Fertility from Education.
- Build a degree-3 regression model predicting Fertility from Education.
- Run anova() on the 3 models and discuss the results.

6.8 Model Fitting and Assumptions

In this section we explore important concepts in machine learning that relate to how well a model fits the data. Overfitting is a common problem in many machine learning algorithms we will learn. And we will discuss the bias-variance tradeoff of various algorithms as we learn them.

6.8.1 Overfitting v. underfitting

The anova() results from the polynomial regression indicate the smallest RSS with the degree 4 model. However, none of the p-values are significant, and it is difficult to draw firm conclusions from such a small set of data points. However, the graph in Figure 6.6 gives us an opportunity to talk about underfitting versus overfitting. The linear degree=1 model probably underfits. In contrast the degree 3 and 4 models might be overfitting the data. When you underfit the data, your model does not have sufficient complexity to explain the data. That is what we see with the degree=1 model. The straight line is not capturing some of the complexity in the data. On the other hand, overfitting is when the model has too much complexity. The principle of **Occam's razor** tells us that when choosing between two likely explanations, choose the simpler one. In this case we might choose degree=2 since it did have a lower p-value. In a scenario where you have a train and test set, if the data performs well on the training data but poorly on the test data you may have overfit. Your model tuned itself too much to variation in the training data and this limited its ability to generalize to new data. On the other hand, if your model does poorly on the training set, you may have underfit. Figure 6.7 illustrates overfitting versus underfitting.

6.8.2 Bias and variance

A common theme throughout this handbook will be the bias-variance tradeoff which is related to underfitting and overfitting. Each algorithm that we learn will have tendencies one way or

[1] Reminder: Readers of the grayscale print book can find the color graphs in the github

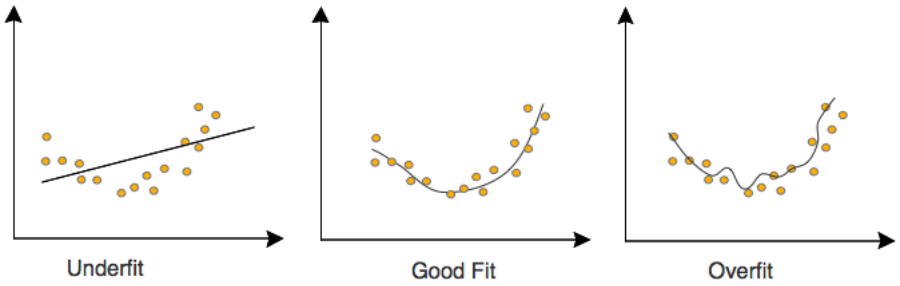


Figure 6.7: Fitting a model

the other. In the underfitting illustration in Figure 6.7, we see that the straight line underfit. It had a strong bias that the data was truly linear and is too simple a model for the data. On the other extreme, the overfitting example showed what can happen if variance is too high: the model learned random things from the data.

So bias-variance and underfitting-overfitting are related but it is important to distinguish their seperate meanings. A high bias, low variance model is likely to underfit and not capture the true shape of the data. This tends to happen more with simpler models like linear regression or logistic regression. In contrast, a low bias, high variance model captures too much complexity and noise in the data and may not generalize well to new data. This can happen with more complex models like decision trees, SVM, or neural networks.

So if you suspect your model has high bias, what can you do? We could try different algorithms. Right now, that's not helpful because we have only learned one algorithm. As we go we will learn more algorithms and learn their tendencies towards either bias or variance. We will see one tool in this chapter however that can help, and that is adding a regularization term that will help linear regression pull back its tendency towards bias. Also in this chapter we learned that a linear model doesn't necessarily mean a straight line. It can be a polynomial line or any mathematical transformation of the linear equation. One more technique we can try to reduce bias is to add more features, if available.

If you suspect your model is suffering from high variance, what can you do? More data should help. Algorithms that tend toward high variance are overly sensitive to noise in the data. Adding more data can quiet the effect of a few noisy observations. Another approach is to try fewer features if you are using a lot of features. Some features may be noiser than others so this could help.

Exercise 6.7 — Interaction Effects. In R, we can add interaction effects to formulas using the * operator. For example, `y~x1+x2+x1*x2` has three predictors: x1, x2, and the interaction of x1 and x2. Again using the swiss data, create a linear model with all predictors plus an interaction between Education and Infant.Mortality. Is this interaction a good predictor? Is the model better? ▪

6.8.3 Linear Model Assumptions

A linear model first and foremost assumes some linear shape in the data. Beyond that, the linear model also has an **additive assumption**, that each predictor contributes to the model independently of the other predictors. In reality, some predictors may be correlated, in which case we might consider removing one of them, since it will be hard for the model to assess the effect of them independently and thus the coefficient estimates may be erroneous. Other predictors may have an **interaction effect**, a synergy between them. Yet another concern is **confounding** variables, which are variables that correlate with both the target and a predictor. How can we detect these situations? We can use the cor() and pairs() methods in R to quantify and visualize correlations in the data set.

6.9 Advanced Topic: Regularization

An extension of the least squares approach is to add a regularization term to the RSS, with its importance controlled by another parameter, lambda. This extra term penalizes large coefficients. When lambda=0 it is the same as the least squares estimate. As lambda gets larger, the coefficients will shrink. The intercept does not shrink because the goal is to reduce the coefficients associated with predictors. The notation $||w||^2$ denotes the *l2* norm, which is $\sqrt{(}\sum_{j=1}^{p})w_j^2$. Regularization can help prevent overfitting when you have relatively complex models on a small data set.

$$\mathbf{RSS} = \sum_{i=1}^{n}(\mathbf{y}_i - \mathbf{w}^T\mathbf{x}_i)^2 + \lambda||\mathbf{w}||^2 \tag{6.27}$$

We can implement ridge regression with the R package glmnet. We will use the airquality data set. Since airquality has a lot of NAs we will omit observations with NAs in the columns we care about with the complete.cases() function. Before performing ridge regression, we build a multiple linear regression model as usual.

Code 6.9.1 — Linear Regression. Multiple Linear Regression on Airquality

```
df <- airquality[complete.cases(airquality[, 1:5]),]
df <- df[,-6]
set.seed(1234)
i <- sample(1:nrow(df), .75*nrow(df), replace=FALSE)
train <- df[i,]
test <- df[-i,]
lm1 <- lm(Ozone~., data=train)
pred <- predict(lm1, newdata=test)
mse1 <- mean((pred-test$Ozone)^2)
```

The output above for the mse is 409.3799. Let's see if we can beat that with ridge regression. First we use the model.matrix() function to create a matrix of the predictors. Then we split into the same train and test observations as for lm1.

Code 6.9.2 — Regularization. Ridge Regression On Airquality

```
library(glmnet)
x <- model.matrix(Ozone~., df)[,-1]
y <- df$Ozone
train_x <- x[i,]
train_y <- y[i]
test_x <- x[-i,]
test_y <- y[-i]

# build a ridge regression model
rm <- glmnet(train_x, train_y, alpha=0)

# use cv to see which lambda is best
set.seed(1)
cv_results <- cv.glmnet(train_x, train_y, alpha=0)
l <- cv_results$lambda.min

# get data for best lambda, which is the 99th
pred2 <- predict(rm, s=l, newx=test_x)
mse2 <- mean((pred2-test_y)^2)
coef2 <- coef(rm)[,99]
```

The mse for the ridge regression was 371.0138, which is about 10% lower than for the regular multiple regression. Let's confirm that the ridge regression shrunk our coefficients. It appears that all the coefficients shrunk a bit.

```
> lm1$coefficients
 (Intercept)      Solar.R         Wind         Temp        Month
-66.85709002   0.08314323  -3.75229006   1.98524049  -3.27749222

> coef2
 (Intercept)      Solar.R         Wind         Temp        Month
-60.80449134   0.08165752  -3.61256523   1.83183505  -2.60738344
```

6.10 Summary

In this chapter we learned the supervised regression technique of linear regression, where our target was a real number variable and our predictors could be any combination of quantitative or qualitative variables. Linear regression has a strong bias in that it assumes that the relationship between the target and the predictors is linear. Keep in mind that *linear* does not always mean a straight line as we saw in the examples.

When we run the linear regression algorithm on training data, we create a *model* of the data that can then be used for predictions on new data. Our model gives us the coefficients

which quantify the effect of each predictor on the target variable. As we explore many more algorithms for regression in the book, we will typically use linear regression as a baseline algorithm to see if other algorithms can beat it. If the data is linear, linear regression is quite hard to beat.

Linear regression strengths and weaknesses:

Strengths:

- Relatively simple algorithm with an intuitive explanation because the coefficients quantify the effect of predictors on the target variable.
- Works well when the data follows a linear pattern.
- Has low variance.

Weaknesses:

- High bias because it assumes a linear shape to the data.

6.10.1 Terminology

This chapter introduces a lot of terminology as we explored simple linear regression, multiple linear regression, and polynomial linear regression. There is a glossary at the end of the book but you might want to read back through the chapter again if you are unsure of the meaning of any of the following terms.

Terms related to the data:

- outlier
- leverage point
- dummy variables
- confounding variables

Terms related to the algorithm:

- coefficients
- residuals
- loss function, or cost function
- gradient descent
- additive assumption of linear regression
- interaction effect in linear predictors

Terms related to metrics:

- correlation
- covariance
- mse mean squared error
- rmse root mean squared error
- rss residual sum of squared errors
- rse residual standard error
- R^2 and adjusted R squared
- F-statistic
- p-value

Terms relevant to all machine learning algorithms:

- overfitting
- underfitting
- bias
- variance
- regularization

6.10.2 Quick Reference

Reference 6.10.1 Create Train and Test Sets

```
set.seed(...)
i <- sample(1:nrow(df), nrow(df)*0.8, replace=FALSE)
train <- df[i,]
test <- df[-i,]
```

Reference 6.10.2 Build and Examine a Linear Regression Model

```
lmName <- lm(formula, data=train)
summary(lmName)
```

Reference 6.10.3 Predict on Test Data

```
# predict using the test data
pred <- predict(lmName, newdata=test)

# predict for a single value
#   where the formula was target~predictor
pred <- predict(lm1, data.frame(predictor=5))
```

Reference 6.10.4 Evaluate Regression Predictions

```
# pred is a vector of real-number predictions
mse1 <- mean((pred - test$y)^2)
cor1 <- cor(pred, test$y)
```

Reference 6.10.5 Comparing Models with anova()

```
anova(model1, model2, ..., modeln)
```

Reference 6.10.6 Building formulas

```
lm1 <- lm(y~., data=train)  # y~. means use all predictors
lm1 <- lm(y~.-x1, data=train)  # y~.-x1 all  except x1
lm1 <- lm(y~x1+x2, data=train)  # y~x1+x2 use x1 and x2
lm1 <- lm(y~x1+x2+x1*x2, data=train)  # x1*x2 interaction effect
lm1 <- lm(y~I(x^2), data=train)  # I() needed for x^2
```

6.10.3 Practice to Consolidate Skills

Problem 6.1 — Practice on the Abalone Data. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Abalone`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Run cor() and pairs() to see if there are any columns you might consider getting rid of.
5. Divide the data into 80-20 train-test, setting a seed for reproducibility.
6. Create a linear regression model with all predictors. What is the correlation of the predicted and actual values? What is the mse?
7. Create at least 2 more models, trying different features and combinations of features to see if you can improve these results.

Problem 6.2 Based on your experience with the linear regression algorithm in R, does removing predictors with low p-values necessarily improve performance? Discuss possible reasons for your answer.

Problem 6.3 If you found that some predictors had high p-values, what reasons might you give for leaving them in? What reasons might you give for taking them out?

6.10.4 Next-Level Learning

There are entire statistics courses devoted to linear regression and scores of books if you have time for a deep exploration. Here are some recommendations:

- Free online linear regression tutorial here: `https://onlinecourses.science.psu.edu/stat501/node/250`
- *Linear Models in Statistics* by Rencher and Schaalje.
- *Linear Models with R* by Faraway.

7. Logistic Regression

7.1 Overview

Despite its name, when we use logistic regression, we are performing **classification**, not regression. Whereas in linear regression, our target variable was a *quantitative* variable, in logistic regression, our target variable is *qualitative*: we want to know what class an observation is in. In the most common classification scenario, the target variable is a binary output so that we classify into one class or the other. As we will see later in the chapter, there are techniques that allow classification into more than two classes.

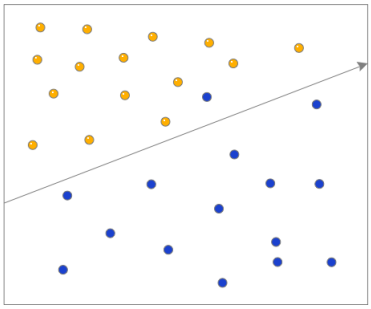


Figure 7.1: Decision Boundary for Binary Classification

Linear models for classification create decision boundaries to separate the observations into regions in which most observations are of the same class. The decision boundary is a linear combination of the X parameters. As we see in Figure 7.1 the two classes are almost perfectly separated by the decision boundary. There is one misclassified observation.

7.2 Logistic Regression in R

Let's take a look at the plasma data set in package HSAUR. This is a blood screening data set for 32 patients that gives measures of two plasma proteins, fibrinogen and globulin, and a binary indicator associated with the two protein levels. The fibrinogen and globulin variables are quantitative. The qualitative variable ESR>20 indicates whether the erythrocyte sedimentation rate, the rate at which red blood cells settle in blood plasma, is over 20 or not. In logistic regression our target needs to be a qualitative variable. In this data set, ESR>20 is our target. It has been coded as a factor in R so that ESR>20 = 2 means that ESR is over 20 and 1 means otherwise. We want to learn to predict whether ESR>20 or not, based on the levels of the plasma proteins fibrinogen and globulin. Values >20 indicate some possible associations with various health conditions.

7.2.1 Plotting factor data

Code 6.2.1 shows how the plots in Figure 7.2 were generated. First, we specify a 1x2 layout for the plots, then use the plot(x, y) command. The parameter `varwidth=TRUE` makes the boxplot widths proportional to the square root of the samples sizes. This easily lets us see that ESR<20 is more common than ESR>20. More importantly, the box plots show that ESR>20 observations are associated with slightly higher levels of globulin and significantly higher levels of fibronogen.

Code 7.2.1 — Plotting Factors. Boxplots.

```
par(mfrow=c(1,2))
plot(ESR, fibrinogen, data=plasma, main="Fibrinogen",
    varwidth=TRUE)
plot(ESR, globulin, data=plasma, main="Globulin", varwidth=TRUE)
```

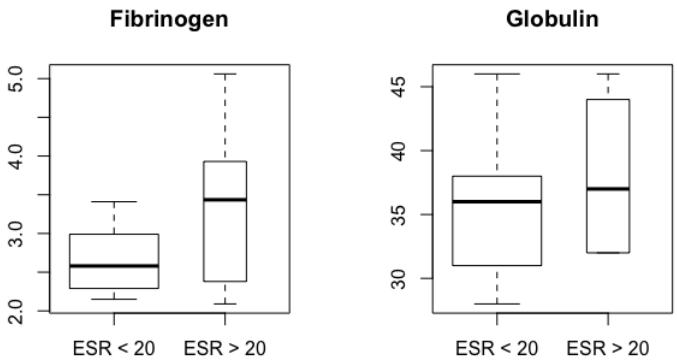


Figure 7.2: Box Plots

Plotting Qualitative Data

If you use the R plot() function with command:

```
plot(globulin, ESR, data=plasma)
```

you will see a row of points at y=1 and another row of points at y=0 on the y axis. This should be your first clue that you need to rethink your plot. A boxplot is created if X is the qualitative variable and Y is the quantitative variable.

In Code 6.2.2 we make two conditional density (CD) plots, shown in Figure 7.3. We can make the same observations as we did when looking at the box plots. Here they are just visualized differently. The total probability space is the rectangle, with the lighter grey indicating ESR>20.

Code 7.2.2 — Plotting Factors. CD Plots.

```
par(mfrow=c(1,2))
cdplot(ESR~fibrinogen)
cdplot(ESR~globulin)
```

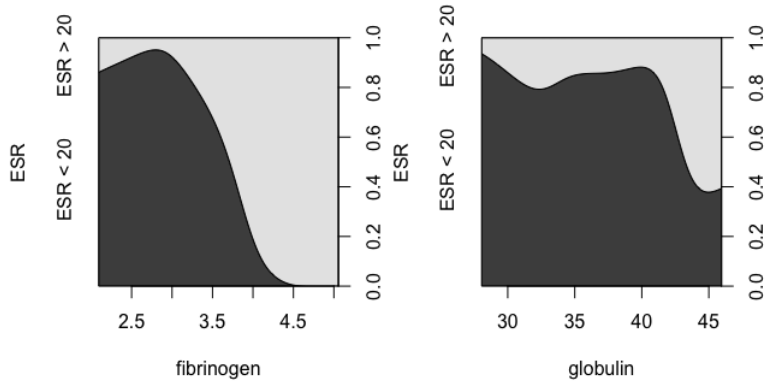


Figure 7.3: Conditional Density Plots

7.2.2 Train and Test on the Plasma Data

Even though this data set is very small, only 32 observations, we divided it into train and test sets, then created a logistic regression model using fibrinogen to predict ESR>20. The full notebook is available online.

For logistic regression we use the glm() *generalized* linear function instead of lm() that we used for linear regression. Also we need the parameter `family=binomial` for logistic regression. Otherwise the glm() function call looks similar to what we have done previously for lm(). The first argument is the formula, which seeks to learn the target ESR with one predictor, fibrinogen.

Code 7.2.3 — Logistic Regression. Using glm().

```
set.seed(1234)
i <- sample(1:nrow(plasma), 0.75*nrow(plasma), replace=FALSE)
train <- plasma[i,]
test <- plasma[-i,]
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
summary(glm1)
```

You will notice that the output of the summary() function is very similar to the output we saw for linear regression, with 4 sections:

- the glm() call
- the residual distribution
- the coefficients with statistical significance metrics
- metrics for the model

Here is the summary output:

```
Call:
glm(formula = ESR ~ fibrinogen, family = binomial, data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-0.9852  -0.7375  -0.5074  -0.1920   2.2554

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -5.6141     2.6591  -2.111   0.0347 *
fibrinogen    1.5084     0.8543   1.766   0.0775 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 26.992  on 23  degrees of freedom
Residual deviance: 22.716  on 22  degrees of freedom
AIC: 26.716

Number of Fisher Scoring iterations: 4
```

7.2.3 Interpreting summary() in Logistical Regression

For the logistic regression output, it is important to note that the residuals are *deviance residuals*. What does that mean? The deviance residual is a mathematical transformation of the loss function (discussed in a later section) and quantifies a given point's contribution to the overall likelihood. These deviance residuals can then be used to form RSS-like statistics.

At the bottom of the output, the statistics section is quite different. We have null deviance and residual deviance for metrics. The null deviance measures the lack of fit of the model, considering only the intercept. The residual deviance measures the lack of fit of the entire model. We want to see that the Residual deviance is much lower than the Null deviance. The AIC is most useful in comparing models, the lower the AIC the better. AIC stands for Akaike Information Criterion and is based on deviance. AIC shows a preference for less complex models with fewer predictors. The Fisher scoring algorithm is a modified form of Newton's method of solving a maximum likelihood problem.

Interpreting coefficients of a logistic regression model is quite different from interpreting them in a linear regression model. Whereas the coefficient of a linear regression predictor quantifies the difference in the target variable as the predictor changes, in logistic regression, the coefficient quantifies the difference in the log odds of the target variable. We will discuss odds, log odds, and probability in the Metrics section.

7.2.4 Evaluation on the Test Data

Next, we look at the output of predict() for logistic regression created in Code 6.2.4. Notice the parameter `type="response"`. This is important to get probabilities out of the model. The model outputs log-odds but by requesting "response" we get these numbers converted to probabilities. The probabilities for the first few test observations are:

```
> head(probs)
         1          3         10         11         22         24
0.14028110 0.09023866 0.15547539 0.10341103 0.16563738 0.10202084
```

The ifelse() function is needed to convert these probabilities to 1 or 2, the internal coding for the target variable. Once the predictions are in variable pred, we can compare them to the actual values in the test observations to get accuracy, the percentage of observations that were classified correctly. The code also outputs a table of predictions and actual values. All the predictions were of class 1. All 8 actual values in this tiny test set were of class 1, giving 100% accuracy. On such a small and unbalanced data set, we should be skeptical about this seemingly good accuracy.

Code 7.2.4 — Logistic Regression. Evaluating the output.

```
probs <- predict(glm1, newdata=test, type="response")
pred <- ifelse(probs>0.5, 2, 1)
acc1 <- mean(pred==as.integer(test$ESR))
print(paste("glm1 accuracy = ", acc1))
table(pred, as.integer(test$ESR))
```

```
[1] "glm1 accuracy =  1"

pred 1
   1 8
```

7.2.5 Learning (or Not) From Data

Note one odd thing about the table above: the model always predicted class 1 (ESR<20). Even though 100% accuracy sounds great, we cannot be impressed. There are two problems here: (1) a small amount of data, (2) an unbalanced data set.

The first problem is that we have a very small data set. In order to create a stronger model, much more data would have to be collected. Data collection can be expensive and require domain expert assistance. In the case of this data set, more blood samples would have to be drawn from a random population, analyzed by technicians, and supervised by a hematologist or other clinician.

When additional data cannot be obtained, another but less preferable option is to use sampling techniques. Consider our example above, with 8 test cases randomly sampled from the data. What if we randomly sampled the data multiple times? By randomly sampling iteratively, each time we would have a different test set. We could average our test accuracy over all these samples and get a better idea of the accuracy of our model. One such sampling technique is cross-validation which we will explore later in the book.

The second problem with the data set is that it is unbalanced. Of the 32 observations, only 6 have ESR>20. This is a 81% ESR<20 to 19% ESR>20 ratio. Unbalanced data sets can pose problems for some classification algorithms while others can rise above it. Again, more data would be helpful. If that additional data is still unbalanced, sampling techniques may be of help. The idea is to oversample from the minority class and undersample from the majority class to come up with a data set that is more balanced. We will explore techniques for doing this in future chapters, and discuss when it might be helpful. For now, we will take our model as created, but take it with a grain of salt.

Exercise 7.1 — Logistic Regression. Practice on the PimaIndiansDiabetes2 data set from package mlbench. First, create an 80-20 split into train and test sets. What is your accuracy predicting diabetes from glucose?

By the way, this data set has a lot of missing NA values. If you get NA for your accuracy, it means that one or more of the predictions was NA. Check this with code: `sum(is.na(pred))`. If you have NAs, compute your mean accuracy using parameter `na.rm=TRUE` in the mean() function. ▪

7.3 Metrics

Classification can be evaluated by many measures. In this section we will look at accuracy, sensitivity and specificity, Kappa, AUC, and ROC curves.

7.3.1 Accuracy, sensitivity and specificity

The most common metric to evaluate results in classification is accuracy:

$$acc = \frac{C}{N} \tag{7.1}$$

where C is the number of correct predictions, and N is the total number of test observations. The output of the table() command above was limited because the model predicted all test cases to be in one class. Normally it should look something like this:

```
pred 1 2
   1 7 1
   2 1 5
```

Such a table is also called a confusion matrix. The diagonal values from the upper left to the lower right are the correctly classified instances, 7 and 5 in this case. The other values are errors, 1 and 1. The flip side of accuracy is the error rate, calculated by subtracting accuracy from 1.

We can break down each component of the confusion matrix as follows:

```
pred T  F
   T TP FP
   F FN TN
```

- TP - true positive: these items are true and were classified as true
- FP - false positive: these items are false but were classified as true
- TN - true negative: these items are false and were classified as false
- FN - false negative: these items are true but were classified as false

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7.2}$$

$$error\ \ rate = \frac{FP + FN}{TP + TN + FP + FN} = 1 - accuracy \tag{7.3}$$

The **sensitivity** measures the true positive rate:

$$sensitivity = \frac{TP}{TP + FN} \tag{7.4}$$

The **specificity** measures the true negative rate:

$$specificity = \frac{TN}{TN + FP} \tag{7.5}$$

Sensitivity and specificity range from 0 to 1, just as accuracy does, with values closer to 1 being better. They help to quantify the extent to which a given class was misclassified.

7.3.2 Kappa

Cohen's Kappa is a statistic that attempts to adjust accuracy by accounting for the possibility of a correct prediction by chance alone. Kappa is often used to quantify agreement between two annotators of data. Here we are quantifying agreement between our predictions and the actual values. Kappa is computed as follows:

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \tag{7.6}$$

where Pr(a) is the actual agreement and Pr(e) is the expected agreement based on the distribution of the classes. The following interpretation of Kappa is often used but there is not universal agreement on this. Kappa scores:

- <2.0 poor agreement
- 0.2 to 0.4 fair agreement
- 0.4 to 0.6 moderate agreement
- 0.6 to 0.8 good agreement
- 0.8 to 1.0 very good agreement

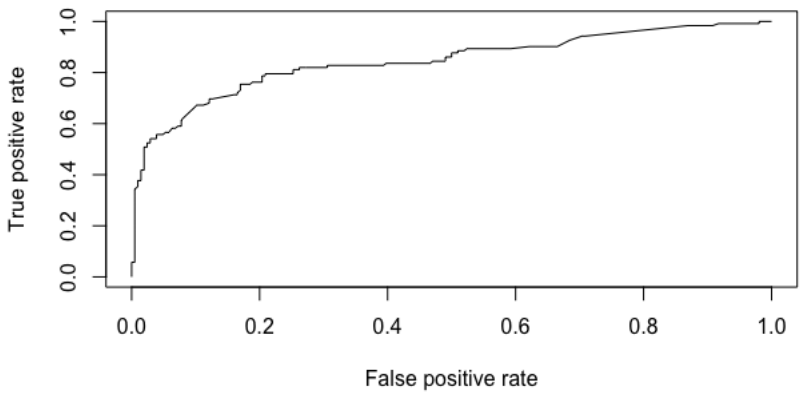


Figure 7.4: ROC Curve

7.3.3 ROC Curves and AUC

The ROC curve is a visualization of the performance of machine learning algorithms. The name ROC stands for Receiver Operating Characteristics which reflects its origin in communication technology in WWII in detecting between true signals and false alarms. The ROC curve shows the tradeoff between predicting true positives while avoiding false positives.

Figure 7.4 shows an ROC curve. This figure is taken from the Titanic logistic regression notebook on the github. The y axis is the true positive rate while the x axis is the false positive rate. The predictions are first sorted according to the estimation probability of the positive

class. A perfect classifier would shoot straight up from the origin since it classified all correctly. We want to see the classifier shoot up and leave little space at the top left. Starting at the orgin, each prediction's impact on the curve is vertical for correct predictions and horizontal for incorrect ones. If we see a diagonal line from the lower left to the upper right, then our classifier had no predictive value.

A related metric is AUC, the area under the curve. AUC values range from 0.5 for a classifier with no predictive value to 1.0 for a perfect classifier.

7.3.4 Probability, odds, and log odds

What is the difference between odds and probability? Let's look at this using a sports outcome example. Imagine we played 10 games with a friend and won 7. That means we lost 3 of course. Assuming we will do as well next time, our odds are 7 to 3:

$$odds = \frac{number\ \ of\ \ wins}{number\ \ of\ \ losses} = \frac{7}{3} \tag{7.7}$$

If we want to express the same data as a probability:

$$probability = \frac{number\ \ of\ \ wins}{number\ \ of\ \ games} = \frac{7}{10} \tag{7.8}$$

Notice that probability will always range from 0 to 1, whereas odds will not. Recall that the glm() algorithm coefficients represent a change in log odds. Just as it sounds, log odds are the log of the odds: log(odds). So to find the odds, we use the inverse of log, the exp() function. And to convert odds to probability, we use the following:

$$probability = \frac{odds}{1 + odds} \tag{7.9}$$

Let's see what this means in terms of the plasma data logistic regression model above. The predictor is fibrinogen, which ranges from around 2.09 to 5.06 in this data set. Let's compare the effect of fibrinogen across values from 2.25 to 5.0. Figure 7.5 plots the log odds across this range of values on the left, and on the right, the associated probabilities across the same range of values. Observe that the results of the logistic regression model (the log odd) are linear in the parameters (w, b) but that the associated probabilities are not linear.

The coefficient for fibrinogen in the linear regression model was 2.34. For every one-unit increase in x, the probability of ESR>20 changes by exp(2.34)/[1+exp(2.34)]. Let's look at some sample values in Table 7.1. The X column is fibrinogen for a range of values. The log odds is found by plugging in the value of x for the logistic regression formula given by: 2.34 * x - 8.383.

As you see in Table 7.1 and more clearly in Figure 7.5, the log odds are linear in the parameters but the probability is not linear, we can discern a subtle S-shape in the probabilities.

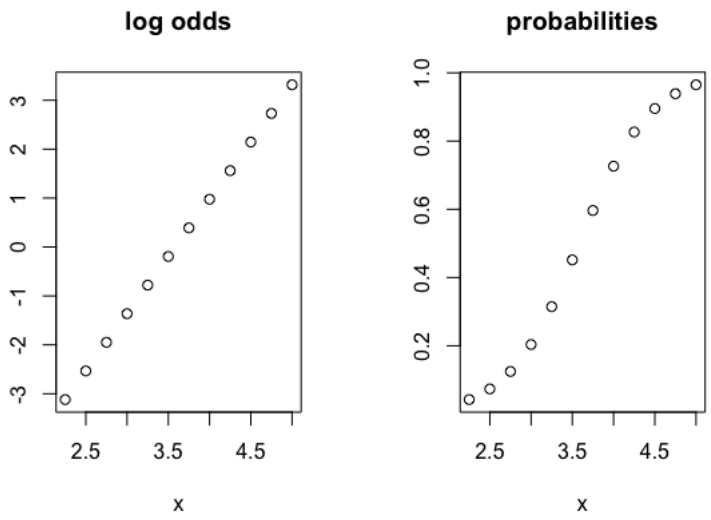


Figure 7.5: Log Odds versus Probability

X	Log Odds	Probability
2.5	-2.53	0.07
3.0	-1.36	0.20
3.5	-0.19	0.45
4.0	0.977	0.73
4.5	2.147	0.89

Table 7.1: Log Odds and Probability for Plasma Data

Logistic Regression Coefficients

In linear regression we interpret a predictor coefficient as the amount of change in y for a 1-unit change in x. We cannot make this intepretation in logistic regression. The predictor coefficient in a logistic regression model specifies the change in log-odds for a 1-unit change in x.

Exercise 7.2 — Metrics for Classification. Using your model created for the Pima Indians data, compute the following in R using the formulas above:

- TP, TN, FP and FN
- accuracy using TP, TN, FP, FN
- error rate using accuracy
- sensitivity
- specificity

Next, use the confusionMatrix() function in package `caret` to confirm your results. ▪

7.4 The Algorithm

The linear regression output was a quantitative value that could range over all the real numbers. What we need for logistic regression classification is a function that will output probabilities in the range [0, 1]. The sigmoid, or logistic, function is used for this purpose and of course is where the algorithm gets its name. When real numbers are input to the logistic function, the output is squashed into the range [0,1] as seen in Figure 7.6. The logistic function is:

$$f(x) = \frac{1}{1+e^{-x}} \tag{7.10}$$

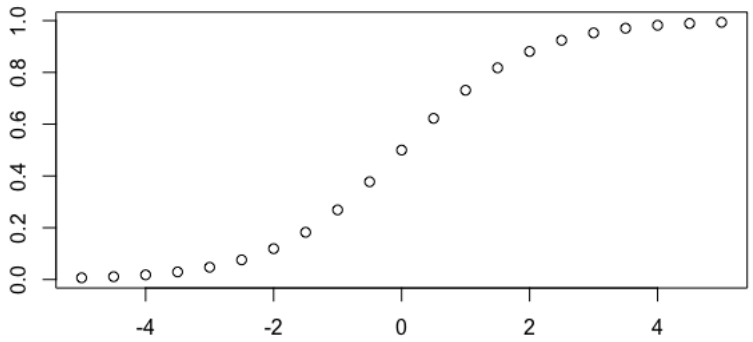


Figure 7.6: The Logistic Function

The logistic regression algorithm computes the log odds from the estimated parameters. The log odds is just log(odds). The odds are the probability of the positive class, p(x), over the probability of the negative class (1 - p(x)). If we have a single predictor w_1 and an intercept w_0, the log odds are:

$$log\frac{p(x)}{1-p(x)} = w_o + w_1 x \tag{7.11}$$

Solving for p gives us the logistic function:

$$p(x) = \frac{e^{-(w_0+w_1 x)}}{1+e^{-(w_0+w_1 x)}} = \frac{1}{1+e^{-(w_0+w_1 x)}} \tag{7.12}$$

We can see in Equation 6.11 why logistic regression is considered a linear classifier. It creates a linear boundary between classes in which the distance from the boundary determines the probability. When we use the logistic function for classification, the cut-off point is usually

0.5. Notice in Figure 7.6 that this is the inflection point of the S-curve. Probabilities greater than 0.5 are classified as the positive class and probabilities less than 0.5 are classified in the other class.

Exercise 7.3 — Logistic Regression Algorithm. Using the coefficients and data from the logistic regression Check-Your-Understanding above, try the following:

- Compute the probabilities in R using only the glucose column from the test data and the coefficients of the model.
- Compare these to the output of predict() you did earlier. Are they the same?
- Create a plot with test glucose on the x axis and the probabilities you calculated for the test set on the y axis. What do you observe?
- Create a vector of x values: 60, 100, 140, 180, 220
- Compute probabilities for these values using the same formula as above.
- Are these probabilities consistent with the graph you just created?

7.5 Mathematical Foundations

How are the parameters, **w**, found for logistic regression? First an appropriate loss function is established, then an optimization technique such as gradient descent is used to find optimal parameters.

Recall the loss function for linear regression:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{7.13}$$

If we plug in the logistic function for f(x), it will not be a convex function. That is a problem because gradient descent works only for convex functions. A suitable loss function for logistic regression can be found by starting with the likelihood function, L:

$$L(w_0, w_1) = \prod_{i=1}^{n} f(x_i)^{y_i}\ (1 - f(x_i))^{1-y_i} \tag{7.14}$$

Notice in the likelihood equation that one of the terms in the product will always reduce to 1 because the y values are either 0 or 1. To simplify the likelihood equation we will take the log of it to find the log-likelihood, ℓ:

$$\ell = \sum_{i=1}^{n} y_i\ log f(x_i) + (1 - y_i)\ log(1 - f(x_i)) \tag{7.15}$$

The log-likelihood equation for each instance:

$$\ell = y\ log f(x) + (1-y)\ log(1-f(x)) \tag{7.16}$$

In training the classifier, we want to penalize it for wrong classifications. This is our loss function. The penalties follow directly from Equation 6.16. Our Loss is:

$$\mathcal{L} = -log(f(x))\ if\ y = 1 \qquad \mathcal{L} = -log(1-f(x))\ if\ y = 0 \tag{7.17}$$

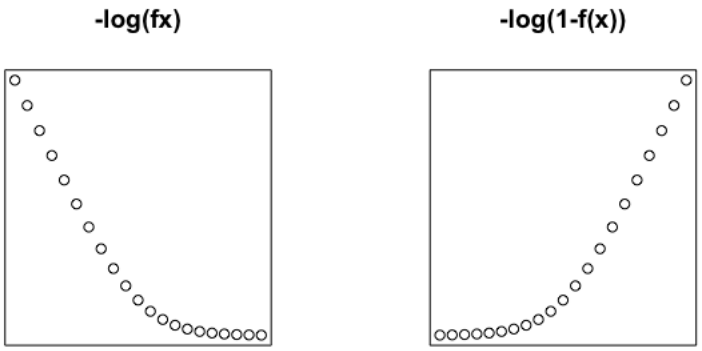


Figure 7.7: Loss Function for Logistic Regression

These two loss functions are visualized in Figure 7.7. In the leftmost graph which represents -log(fx), the closer the function gets to 1, the smaller the penalty but the closer it gets to 0, the higher. We want to penalize -log(f(x)) severely if it classifies as 0 when the true target is 1. The reverse is true when the true target is 0. This is shown in the rightmost graph. Here we penalize -log(1-f(x)) severely as it moves toward 1 because the true target is 0.

We can put the two loss functions into one equation as shown below, where they are summed over all observations. Notice that one of the terms will always be zero because y will be either 0 or 1.

$$\mathcal{L} = -\Big[\sum_{i=i}^{N} y_i log(f(x_i)) \quad + \quad (1-y_i) log(1-f(x_i))\Big] \tag{7.18}$$

where f(x) =

$$f(x) = \frac{1}{1+e^{-(w^T x)}} \tag{7.19}$$

The parameters w are then determined using gradient descent. You can see from Figure 7.7 that when you put together the two loss functions you will have a convex function suitable for gradient descent. Once the parameters are known, classifying new instances is done by plugging in the instance into the logistic function shown in Equation 6.19. This returns a probability in the range [0, 1]. Typically we establish a threshold such as 0.5. Values over that threshold are classified as 1, values below are classified as 0.

Likelihood versus Probability

In common speech we use the terms likelihood and probability interchangeably, but in statistics they have different meanings. Likelihood is looking backward in time. Likelihood is the probability that an event which already happened would yield a specific outcome. Probability looks forward in time. Probability is estimating a future outcome given known values. Probabilities for all possible values sum to 1. Likelihoods do not have this limit. We'll discuss this more in the next chapter.

7.6 Logistic Regression with Multiple Predictors

The github repo for the book has a logistic regression example on a Titanic data set. We will just cover a few points here. The first is that the data had a lot of NA values. NA values can cause a lot of problems for many classifiers and it is a good idea to remove them. There are a couple of approaches to take. One is to simple remove rows with NAs which is a good choice if you have lots of remaining data. We saw an example of that using R's complete.cases() function in the linear regression chapter. Another approach is to replace NA values with either the mean or median of the variable for quantitative variables, or the most common factor for qualitative variables. The code below shows the detection of NAs with the sapply() function.

Code 7.6.1 — Detecing NAs. Using sapply().

```
sapply(df, function(x) sum(is.na(x)==TRUE))
  pclass survived      sex      age
       0        0        0      263
```

The output shows that almost a third of the examples in the data set have missing values for age. Getting rid of these rows would leave a much smaller data set. Instead, we replace NAs with the median value. That is shown in the code below. The median() function needs the argument 'na.rm=T' in order to compute the median of the values that are not NA.

Code 7.6.2 — Removing NAs. By deleting rows or replacing with the median.

```
df$age[is.na(df$age)] <- median(df$age,na.rm=T)
```

One concern about replacing age NAs with median values is that there were so many NAs, almost one-third of the data. By replacing so many NAs with the mean or median, we are diminishing the predictive power of age. An alternative is to replace age NAs for observations

that survived with the mean or median for age of those who survived, and then do something similar for NAs for observations that did not survive. This approach is somewhat problematic as well. We are manipulating our data. Whatever decisions you make should be thoroughly documented in your RStudio notebook and any other reporting you do on your results.

In the online notebook you will see that we divided the data into train and test sets and build a logistic regression model which achieved 76.5% accuracy. The code also gives an example of using the `caret` package to output the confusion matrix and other statistics including the Kappa. The ROC curve for our predictions is the ROC curve shown in the Metrics section above. The AUC for that curve was 0.81. The following is the key output of summary() for the model predicting survival with all predictors.

```
Call:
glm(formula = survived ~ ., family = "binomial", data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.9509  -0.6567  -0.4336   0.6703   2.4834

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.858516   0.360478  10.704  < 2e-16 ***
pclass2     -1.417739   0.249787  -5.676 1.38e-08 ***
pclass3     -2.437512   0.233637 -10.433  < 2e-16 ***
sexmale     -2.552619   0.175795 -14.520  < 2e-16 ***
age         -0.042339   0.007198  -5.882 4.06e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1305.05  on 980  degrees of freedom
Residual deviance:  897.03  on 976  degrees of freedom
AIC: 907.03

Number of Fisher Scoring iterations: 4
```

Consider the summary shown above. All of the predictors seem to be good predictors. Notice that pclass has dummy variables for class 2 and class 3. How do we interpret this? If a person is pclass 2 instead of 1, the log odds of their survival decreases by 1.4 and if they are pclass 3 instead of 1, the log odds of their survival decreases by 2.4. A similar interpretation can be made for male over female. In this example, age is our only quantitative predictor. The coefficient is telling us that log odds of survival decrease a little for every year. Notice also the large drop in deviance from the null deviance which considers the intercept alone, and the

residual deviance, which considers all predictors. This drop indicates that our predictors are good predictors.

Below we have the code for generating the ROC plot and computing AUC. The AUC is .81.

Code 7.6.3 — ROC and AUC. On the Titanic Data.

```
library(ROCR)
p <- predict(glm1, newdata=test, type="response")
pr <- prediction(p, test$survived)
# TPR = sensitivity, FPR=specificity
prf <- performance(pr, measure = "tpr", x.measure = "fpr")
plot(prf)
# compute AUC
auc <- performance(pr, measure = "auc")
auc <- auc@y.values[[1]]
```

Exercise 7.4 — Logistic Regression with Multiple Predictors. In this example, you will build a logistic regression model using multiple predictors on the Pima Indian data. Try the following:

- Build a model with all predictors, using the same train and test set as you used with only glucose as a predictor. Is your accuracy higher or lower?
- Use R functions to count how many NAs are in each column.
- Replace NAs in the triceps and insulin columns with the mean of each column in both train and test. Now replace train and test with rows that have complete data using complete.cases().
- Build a second model with all predictors on the train and test that have been cleaned up.
- Compare the accuracies of the two models.
- Compare the residual deviance and degrees of freedom for each model. Which model do you think is best, and why?

■

7.7 Advanced Topic: Optimization Methods

Earlier we specified our log-loss function for logistic regression as follows:

$$\ell = \sum_{i=1}^{n} y_i \, log f(x_i) + (1 - y_i) \, log(1 - f(x_i)) \tag{7.20}$$

where f(x) =

$$f(x) = \frac{1}{1 + e^{-(w^T x)}} \tag{7.21}$$

Next we find the gradient of the log likelihood. The gradient, g, is the partial derivative with respect to the parameters **w**. The gradient is the slope which is going to tell the algorithm which direction to move to find the minimum. The gradient equation is given below. Notice that it is really the X matrix multiplied by how wrong f(x) is at this point in predicting the true y.

$$g = \frac{\partial \ell}{\partial \mathbf{w}} = \sum_i (f(x_i) - y_i)\mathbf{x}_i = \mathbf{X}^T(f(x) - \mathbf{y}) \tag{7.22}$$

All we need for gradient descent is the first derivative, the gradient. For other optimization methods we will also need the second derivative, the Hessian, H. The Hessian is a square matrix of second partial derivatives that gives the local curvature of a function. Note that in the following equation for the Hessian, the S represents $S \triangleq diag(p_i(1-p_i))$

$$H = \frac{\partial}{\partial \mathbf{w}} g(\mathbf{w})^T = \sum_i (\nabla_w f(x_i))\mathbf{x}_i^T = \sum_i f(x_i)\,(1 - f(x_i)\,\mathbf{x}_i\,\mathbf{x}_i^T = \mathbf{X}^T\mathbf{S}\mathbf{X} \tag{7.23}$$

There are many algorithms to find the optimal parameters, **w**. The first one we examine is gradient descent.

7.7.1 Gradient Descent

Gradient descent is an iterative approach where at each step the estminated parameters, θ get closer to the optimal values. We can express this as follows:

$$\theta_{k+1} \;=\; \theta_k - \eta_k\, g_k \tag{7.24}$$

where η is the learning rate, which specifies how big of a step to take at each iteration. If the eta (step size) is too slow the algorithm will take a long time to converge. If eta is too large, the true minimum can be stepped over and then the algorithm will not be able to converge.

7.7.2 Stochastic Gradient Descent

For large data sets, gradient descent can bog down. An alternative is *stochastic gradient descent* which processes the data either one at a time or in small batches instead of all at once. It is stochastic because the observations are chosen randomly. If the data is processed one at a time it means that the gradient has to be computed at each step. It turns out that the gradient will reflect the underlying function better if it is computed from a small batch. This also improves computation time.

7.7.3 Newton's Method

Gradient descent finds the optimal parameters using the first derivative. Newton's method is another approach; it uses the second derivative, the Hessian defined above. Newton's method

(also called Newton-Raphson) is also an iterative method. At each step either the full Hessian is recalculated, or it is updated in which we call the method quasi-Newton. In a well-behaved convex function, Newton's method will converge faster.

A key insight in Newton's method is that if it is computationally difficult to compute a minimum for a given function, then come up with a function that shares important properties with the original function but is easier to minimize. At each iteration, Newton's method constructs a quadratic approximation to the objective function in which the first and second derivatives are the same. The approximate function is minimized instead of the original.

How is this approximate function found? A Taylor series about the point is used, but ignores derivatives past the second. A Taylor series converts a function into a power function and the first few terms can be used to get an approximate value for a function.

7.7.4 Optimization from Scratch

We are going to take a closer look at gradient descent by finding our optimal coefficients in the plasma data set from scratch. The R Notebook for this is available online. First we recreate the logistic regression model from earlier in the chapter. Our coefficients were $w_0 = -5.6141$ and $w_1 = 1.5084$.

The first thing we need to do is define our sigmoid/logistic function that will take an input matrix and return a vector of sigmoid values for each observation. We initialize w_0 and w_1 to 1. We make a data matrix where column 1 is all 1s that will be multiplied by the intercept, and column 2 is fibrinogen which will be multiplied by w_1. We will also need the labels but since they were coded as 1-2 instead of 0-1 we subtract 1.

Code 7.7.1 — Logistic Regression from Scratch. Set up code.

```
sigmoid <- function(z){
  1.0 / (1+exp(-z))
}
# set up weight vector, label vector, and data matrix
weights <- c(1, 1)
data_matrix <- cbind(rep(1, nrow(train)), train$fibrinogen)
labels <- as.integer(train$ESR) - 1
```

Now we are ready to iterate. The code below iterates 500,000 times. In each iteration it does the following:

1. multiplies the data by the weights to get the log likelihood, then runs these values through the sigmoid() function to get a vector of probabilities
2. computes the error: the true values (0 or 1) minus the probabilities
3. updates the weights by weights plus the learning rate times the gradient; Recall that the gradient is the X values times the errors as shown in Equation 6.22; Notice also the operator for matrix multiplication is an asterisk surrounded by percent signs.

Code 7.7.2 — Gradient Descent from Scratch. Three steps per iteration.

```
learning_rate <- 0.001
for (i in 1:500000){
  prob_vector <- sigmoid(data_matrix %*% weights)
  error <- labels - prob_vector
  weights <- weights + learning_rate * t(data_matrix) %*% error
}
weights
```

Try running this code several times, changing the number of iterations. The following table shows the weights af various numbers of iterations.

No. Iterations	(w_0)	(w_1)
50	0.483	-0.328
500	-0.097	-0.267
5000	-3.26	0.763
50000	-5.61	1.50
500000	-5.6141	1.5084

Table 7.2: Optimized Weights by Number of Iterations

The 500,000 iterations gives use the same coefficients as R's glm(). However, it was slow compared to R's optimized code. R functions are heavily optimized and will reliably give good performance.

Finally, we create a plot that confirms Equation 6.11. The linear combination of the weights we calculated in the code above and X values give us the log odds.

Code 7.7.3 — Log Odds. Linear combination of $w_0 + w_1 x$

```
plasma_log_odds <- cbind(rep(1, 32), plasma$fibrinogen) %*% weights
plot(plasma$fibrinogen, plasma_log_odds, col=plasma$ESR)
abline(weights[1], weights[2])
```

7.8 Multiclass Classification

The classification examples we have seen so far have been binary, classifying into one of two possible classes. What if we want to classify in a scenario where there are more than two classes? One technique that can be used is **one-versus-all** classification in which we perform multiple binary classifications. Let's look at the iris data set as an example. The notebook for this is in github as usual. The iris data set contains 150 observations of flower measurements. There are 50 observations each of 3 species: virginica, setosa, and versicolor. First we look at a couple of graphs for data exploration. Figure 7.9 shows the pairs() output for the predictor

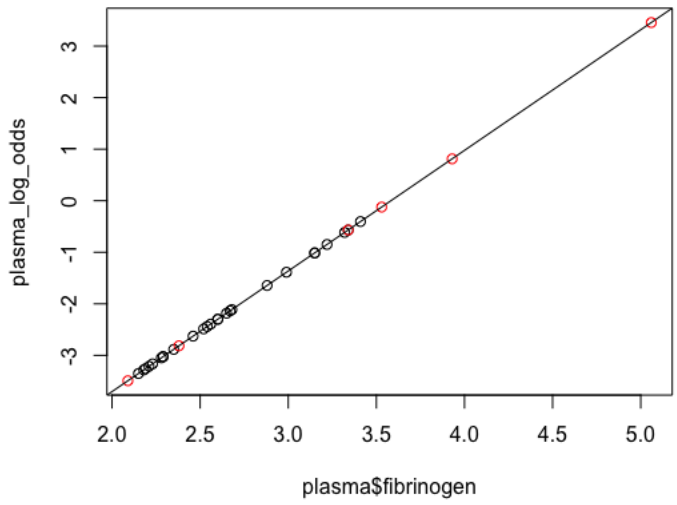


Figure 7.8: LogOdds is a Linear Combination of the Parameters

columns with the color of each observation representing one of the 3 classes. Figure 7.10 plots petal width and length, color coded as well. The code for these plots is given below. The as.integer() function was used to make Species an integer 1, 2, or 3, which in turn is used to match colors red, yellow and blue.

Code 7.8.1 — Code to Generate Plots. Using as.integer()

```
pairs(iris[1:4], pch = 21,
     bg = c("red", "yellow", "blue")[as.integer(Species)])
plot(Petal.Length, Petal.Width, pch=21,
    bg=c("red","yellow","blue")as.integer(Species)])
```

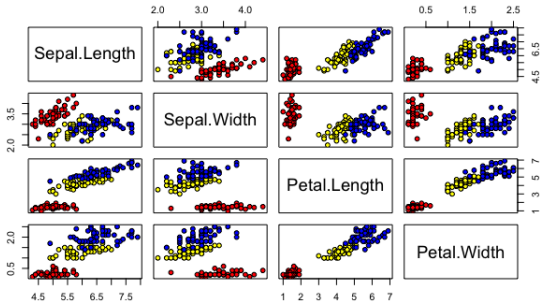


Figure 7.9: Iris Pairs

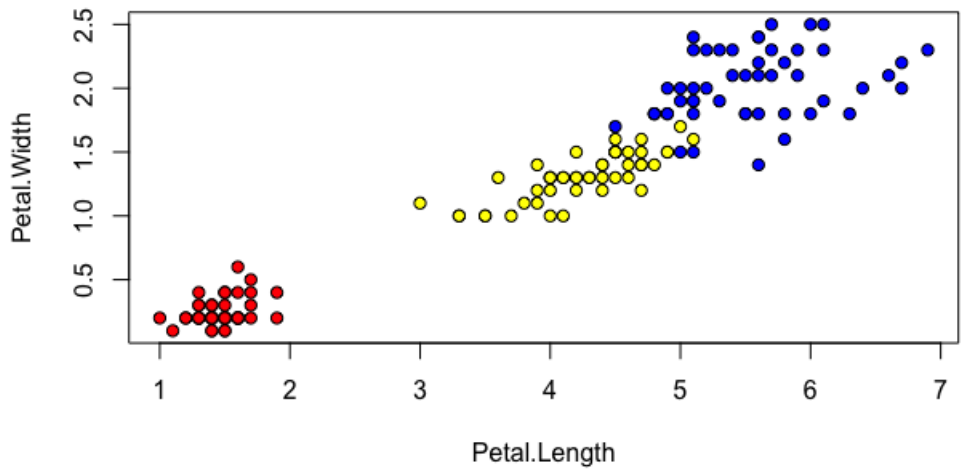


Figure 7.10: Iris Petal Length and Width

To perform one-versus-all classification for 3 classes, we have to create 3 data sets, one for each class as shown next. Each time we copy the original data set, then set the Species column to be a binary factor for that species or not.

Code 7.8.2 — Make a data set for each class. With a binary target.

```
# recode as virginica or not
iris_virginica <- iris
iris_virginica$Species <-
    as.factor(ifelse (iris_virginica$Species=="virginica",1,0))

# recode as setosa or not
iris_setosa <- iris
iris_setosa$Species <-
    as.factor(ifelse (iris_setosa$Species=="setosa",1,0))

# recode as versicolor or not
iris_versicolor <- iris
iris_versicolor$Species <-
    as.factor(ifelse (iris_versicolor$Species=="versicolor",1,0))
```

We next write a function to enable us to run the same code 3 times.

Code 7.8.3 — Function Definition. For repeated code.

```
fun <- function(df, i){
    train <- df[i,]
    test <- df[-i,]
    glm1 <- glm(Species~., data=train, family="binomial")
    probs <- predict(glm1, newdata=test)
    pred <- ifelse(probs>0.5, 1, 0)
    acc <- mean(pred==test$Species)
    print(paste("accuracy = ", acc))
    table(pred, test$Species)
}
```

Next we make one set of indices to divide the train and test sets, and run the function on each data set.

Code 7.8.4 — Run the function. On each data set.

```
set.seed(1234)
i <- sample(1:150, 100, replace=FALSE)
fun(iris_virginica, i)
fun(iris_setosa, i)
fun(iris_versicolor, i)
```

The accuracies for the 3 runs were: 0.98, 1.0, and 0.62. The average gives 87% overall accuracy. From the figures above it is clear that separating setosa (yellow) and versicolor (blue) is challenging and it looks like the versicolor classifier is the weakest one.

How well does this one-versus-all approach scale up? You can imagine that it would be troublesome for classifying 10 classes, as in digit recognition. You would have to build 10 classifiers. As we will see in the deep learning chapter, neural networks can be built to output probabilities for 10 classes, as in this problem scenario. The kNN algorithm is another example of an algorithm that can handle multi-class problems.

Warnings in glm()

For logistic regression, if your training data is perfectly or nearly perfectly linearly separable, R will throw out several warning messages. This is due to the inability to maximize the likelihood which already has separated the data perfectly. For example, for the iris data which is too easy to classify, the sample notebook on github shows the error messages:

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Since there are just warnings, they do not stop the algorithm from producing a model.

7.9 Generalized Linear Models

Logistic regression is part of the GLM (generalized linear model) family because its response is determined by a predictor in which the parameters are linear. In R's glm() function we used the family parameter to specify the family and link function.

```
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
```

For binomial, the link function is "logit". Other available families include gaussian, gamma, poisson and more, as indicated in the R documentation.

Why do we need a link function? In linear regression the algorithm assumes that the target variable is normally distributed over the real numbers. This is not the case for logistic regression and other generalized linear models. The link function links the mean of the target $\mu_i = E(Y_i)$ to the linear term $x_i^T w$. The link function "links" the linear predictors to the response. The canonical link for mapping real numbers to [0,1] is the logit.

7.10 Summary

Logistic regression is something of a misnomer because we use it for classification, not regression. It is considered a linear model because it is linear in the parameters. The sigmoid function shapes the output to be in range [0, 1] for probabilities. Here are the stengths and weaknesses of logistic regression:

Strengths:

- Separates classes well if they are linearly separable
- Computationally inexpensive
- Nice probabilistic output

Weaknesses:

- Prone to underfitting; Not flexible enough to capture complex non-linear decision boundaries

In this chapter we showed how to perform logistic regression with one predictor or multiple predictors. In addition, we discussed how to use the one-versus-all technique for multi-class classification.

7.10.1 New Terminology in this Chapter

Refer back to the chapter or look at the glossary if you are unsure of the meaning of these terms.

New metrics:

- accuracy
- error rate
- sensitivity
- specificity
- Kappa
- ROC Curves
- AUC

Mathematical terms:
- probability
- odds
- log odds

7.10.2 Quick Reference

Reference 7.10.1 Build and Examine a Logistic Regression Model

```
glmName <- glm(formula, data=train, family=binomial)
summary(glmName)
```

Reference 7.10.2 Predict on Test Data

```
probs <- predict(glmName, newdata=test, type="response")
pred <- ifelse(probs>0.5, 1, 0)
table(pred, test$target)
acc <- mean(pred==test$target)
```

Reference 7.10.3 Confusion Matrix

```
library(caret)
confusionMatrix(predictions, test$target)
```

Reference 7.10.4 ROC and AUC

```
library(ROCR)
p <- predict(glm1, newdata=test, type="response")
pr <- prediction(p, test$survived)
# TPR = sensitivity, FPR=specificity
prf <- performance(pr, measure = "tpr", x.measure = "fpr")
plot(prf)
# compute AUC
auc <- performance(pr, measure = "auc")
auc <- auc@y.values[[1]]
```

7.10.3 Practice to Consolidate Skills

Problem 7.1 — Practice on the Abalone Data. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Abalone`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Examine the rings column with range(), median(), and hist() to determine where you would like to split the data into two classes: large and small.
5. Create a new factor column for binary large/small based on the rings column and your cut-off decision.
6. Divide the data into 80-20 train-test, setting a seed for reproducibility.
7. Create a logistic regression model with all predictors except rings. What is the accuracy of the model? Do you think this is a good model? Why or why not?
8. Create at least 2 more models, trying different features, and combinations of features to see if you can improve these results.

Problem 7.2 — Practice on the Heart Data. Try the following:

1. Download the Heart data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Heart+Disease`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Make sure that columns are of the correct type: factors, integer, etc.
4. Check if there are missing values, and how many per column.
5. Divide the data into 80-20 train-test, setting a seed for reproducibility.
6. Create a logistic regression model with all predictors. What is the accuracy of the model? Do you think this is a good model? Why or why not?
7. Create at least one more model, trying different features, and combinations of features to see if you can improve these results.

Problem 7.3 Based on your experience with the logistic regression algorithm in R, does removing predictors with low p-values necessarily improve performance? Discuss possible reasons for your answer.

Problem 7.4 If you found that some predictors were not adding to the performance of the model, what reasons might you give for leaving them in? What reasons might you give for taking them out?

7.10.4 Next-Level Learning

A free online lesson on Logistic Regression is available here: `https://onlinecourses.science.psu.edu/stat504/node/149`

A good discussion of the AIC metric is available here: `https://www.r-bloggers.com/how-do-i-interpret-the-aic/`

8. Naive Bayes

8.1 Overview

Naive Bayes is a popular classification algorithm. The mathematical foundations of Naive Bayes go back to the 18th Century and the mathematician and minister, Thomas Bayes, who formalized this probabilistic equation that bears his name. Bayes theorem:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad aka: \quad \textbf{posterior} = \frac{\textbf{likelihood} \times \textbf{prior}}{\textbf{marginal}} \tag{8.1}$$

Let's consider the above equation in terms of the Titanic data.

$$P(survived|data) = \frac{P(data|survived)P(survived)}{P(data)} \tag{8.2}$$

The quantity $P(data|survived)$ is called the **likelihood**. It is calculated from the training data by determining the joint probabilities of Survived and the Data. It quantifies how likely it is that we would see the data given the Survived instances. The quantity $P(survived)$ is called the **prior** and the distribution of this data is also learned from the training set. The denominator $P(data)$ is used to normalize the fraction to a probability in the range 0 to 1. It is also called the marginal. The quantity to the left of the equal sign is called the **posterior** and it will be the probability of the positive class for a given observation.

8.2 Naive Bayes in R

We will apply the naive Bayes algorithm to the same Titanic data set as we applied logistic regression in the previous chapter. The same steps were used for data cleaning and creating the train/test split, so we will skip those here. In the code below we see that we first load package e1071 which contains the naiveBayes() function. Then we call Naive Bayes on the training data using a formula the same as other algorithms we have used. When we type the name of the model we have built, the information below the code is output.

Code 8.2.1 — Naive Bayes. Requires package e1071.

```
library(e1071)
nb1 <- naiveBayes(survived~., data=train)
nb1
```

```
Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
       0        1
0.617737 0.382263

Conditional probabilities:
   pclass
Y           1         2         3
  0 0.1468647 0.1930693 0.6600660
  1 0.3946667 0.2426667 0.3626667

   sex
Y      female      male
  0 0.1584158 0.8415842
  1 0.6773333 0.3226667

   age
Y       [,1]     [,2]
  0 30.32109 12.32909
  1 28.14467 13.83251
```

Earlier we stated that the prior and likelihood data was calculated from the training set. The output above shows this. The prior for Survived, called A-priori above, is .617 not-Survived and .382 Survived. The likelihood data is shown in the output as conditional probabilities. For discrete variables, this is a breakdown by survived/not for each possible value of the attribute.

For continuous data like age we are given the mean and standard deviation for the two classes. Notice also the reference at the top of the output about laplace. More about that in a later section.

In the output above there are 2 discrete variables, pclass and sex. Notice that each row sums to 1, indicating that they are probabilities. The probabilities of survival=1 for the 3 classes are 39%, 24%, and 36% respectively. The probability of surviving for females was 67.7% compared to 32% for males. The age variable is continuous. The mean for not surviving is 30 and for surviving is 28. These values are very close, so just looking at age alone does not tell us much. However, when we preprocessed this data (see the online notebook) we had a lot of NAs for age that were replaced with the median of age. This could have muddied the waters.

We calculated accuracy on the test set after predicting with parameter type="class". The accuracy for naive Bayes was slightly higher than for logistic regression. Note that we can extract the raw probabilities from the predictions using type="raw".

Code 8.2.2 — Raw Probabilities. With type="raw"

```
p2_raw <- predict(nb1, newdata=test, type="raw")
head(p2_raw, n=2)
```

```
              0         1
[1,] 0.06305836 0.9369416
[2,] 0.12856023 0.8714398
```

Exercise 8.1 — Pima Diabetes Data. Try the following:

- Load the PimaIndiansDiabetes2 data set from package mlbench into variable df.
- Use str() to familiarize yourself with the data and attach the data.
- Use sapply() to get counts of NAs per column.
- Fix the NAs as follows. Replace NAs in triceps and insulin with the mean value of the column. Then remove final NAs with complete.cases().
- Divide the data into a 80/20 train/test split using seed 1234.
- Compare the results of a logistic regression and naive Bayes model.
- Evaluate on the test data. What is your accuracy?

This data set had extremely large numbers of NAs for insulin and triceps, almost a third to one half of the data. By replacing these NAs with the mean of each column we could have diminished their predictive power. What if instead of replacing with the mean, we replace with the class-conditional mean? First we check to see if there is much difference in the means when diabetes is positive versus negative:

```
n <- which(df$diabetes=="pos")
mean(df$insulin[n], na.rm=TRUE)  # 206.8
mean(df$insulin[-n], na.rm=TRUE) # 130
```

Yes, there is. Let's run the algorithms again on updated data:

- Reload the data into df and redivide into train and test using the same i as before.
- Replace NAs for insulin and triceps with the class conditional means. An example is shown below of one of the four lines you will need.
- Run logistic regression and naive Bayes on this updated data.
- Were your results significantly different? Which models do you prefer and why?

```
# Replace NAs on triceps with class-conditional means:
df$triceps[which(df$diabetes=="pos" & is.na(df$triceps))] <-
        mean(df$triceps[n], na.rm=TRUE)
```

8.3 Probability Foundations

This book assumes that readers have had a prior course on probability but we will review some key concepts here. Random variables, often denoted by capital letters such as X, can be discrete or continuous. The probability of two variables X and Y is called the *joint* distribution, determined jointly by X and Y, $P(X, Y)$. The *conditional* distribution of P(X|Y) is given by:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} \tag{8.3}$$

Two important probability rules are the product rule and the sum rule. The product rule says that the joint probability of A and B can be calculated by multiplying the conditional probability of A given B by the probability of B.

$$p(A,B) = p(A|B)p(B) \tag{8.4}$$

The sum rule says that we can calculate the probability of A by finding the joint probability with B and summing over all possible values of b.

$$p(A) = \sum_b p(A,B) = \sum_b p(A|B=b)P(B=b) \tag{8.5}$$

The chain rule lets us take the joint probability of many variables as follows:

$$p(X_{1:D}) = p(X_1)p(X_2|X_1)p(X_3|X_2,X_1)...p(X_D|X_{1:D-1}) \tag{8.6}$$

The expectation of a random variable is also known as the mean, or first moment. The expectation of a discrete random variable is:

$$E(X) = \sum_i X_i P(X_i) \tag{8.7}$$

So the expected value of a fair die is 3.5:

$$E(X) = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = 3.5 \tag{8.8}$$

The variance of a distribution is also called its second moment, and is represented by σ^2. When we take it's square root, we have the standard deviation.

$$\sigma^2 = E[(X - \mu)^2] \tag{8.9}$$

The log trick is often used when multiplying probabilities to prevent underflow and possibly multiplying by 0. Instead of multiplying the probabilities, the log trick says to add the log of the probabilities.

8.4 Probability Distributions

There are a few probability distributions that occur frequently in Bayesian approaches so it would be a good idea to review them here. Many of these are discussed also in context of their conjugate prior. Conjugate distributions are in the same family.

8.4.1 Bernoulli, Binomial, and Beta Distributions

These distributions concern binary variables representing such things as the flip of a coin, wins and losses, and so forth. Our example will be shooting baskets for practice, where x=1 means that we made a basket and x=0 means that we missed. Let's say that I am shooting hoops for the first time and I made 2 baskets out of 10 tries. Given this data, my probability of making a basket is 0.2. The Bernoulli distribution describes binary outcomes like this example. The Bernoulli distribution has a parameter, μ, which specifies the average probability of the positive class.

$$Bernoulli(x|\mu) = \mu^x (1 - \mu)^{1-x} \tag{8.10}$$

This gives us the probability that x is 1: $p(x = 1) = 0.2^1 * .8^0 = 0.2$. And the probability that x is 0: $p(x = 0) = 0.2^0 * .8^1 = 0.8$.

The Bernoulli distribution is a special case of the binomial distribution in which the number of trials, N = 1. Now suppose we run our Bernoulli trial N=100 times, that is, I shoot 100 baskets. Let X be the random variable which represents the number of baskets made. The binomial distribution of X where I made k baskets in N trials has the following probability mass function (pmf):

$$Binomial(k|N,\mu) = \left(\frac{N}{k}\right)\mu^k(1-\mu)^{N-k} \tag{8.11}$$

Let's let k=20 for our 100 trials. Will the outcome of the binomial be 0.2?

$$Binomial(20|100,0.2) = \left(\frac{100}{20}\right)0.2^{20}(1-0.2)^{80} = 0.09930021 \tag{8.12}$$

No, it is not because there are many ways we can get 20 baskets out of 100 trials, 100 choose 20, to be exact. You can get out your calculator to confirm that or use R command `dbinom(20, 100, 0.2)`.

What is the expected mean of our 100 trials? Our expected value will be $N\mu$ which in our case is $100 * 0.2 = 20$. Let's derive these values using R. First we make a vector of possible values for k, the number of baskets made. We could make anywhere from 0 to 100 baskets. Then we multiply each k by its probability and add them together following Equation 6.7 above for the mean of a discrete distribution. E is 20, as we expected. The plot in Figure 8.1 shows the expected value at the center with the variance, calculated as $N\mu(1-\mu)$, which is 16 in our example. If you `sum(dbinom(k, 100, 0.2))` you get 1.0 of course because this represents the total probability space.

Code 8.4.1 — Basketball. A Binomial Distribution.

```
k <- 0:100  # possible number of baskets for 100 tries
E <- sum(k * dbinom(k, 100, 0.2))
v <- 100 * .2 * .8
plot(k, dbinom(k, 100, 0.2))
```

Now suppose that I got really lucky when I shot my first 3 hoops and made all 3 baskets. This gives me $\mu = 1.0$. It's unlikely I can keep this probability over the long haul. In fact, small sample sizes serve poorly as prior estimates of probabilities. Instead of a prior μ we really need a prior distribution over μ. We want this prior distribution to be a *conjugate* of our binomial distribution, meaning that we want it to be proportional to $\mu^x(1-\mu)^{1-x}$. The beta distribution is a good choice:

$$Beta(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{8.13}$$

In the above equation, the first term with the gammas serves as a normalizing constant to make sure that the total probability integrates to 1. The gamma function is commonly used in probability, and is an extension of the factorial function to the real numbers: $\Gamma(n) = (n-1)!$ and is also extended to complex numbers. The parameters a and b in our example will be the number of baskets made and missed, respectively. Beta distributions are commonly used in Bayesian approaches to represent prior knowledge of a parameter. The gamma function is defined as follows for positive real numbers or complex numbers with a positive real portion:

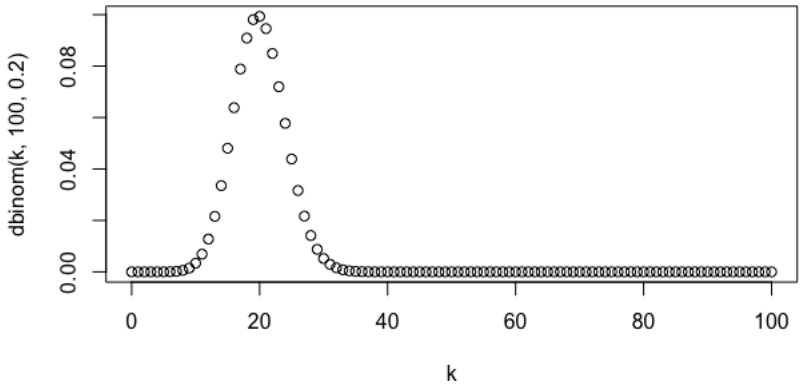


Figure 8.1: Binomial Distribution for 0.2

$$\Gamma(x) \equiv \int_0^\infty u^{x-1} e^{-u} du \tag{8.14}$$

Let's look at the beta distribution for our example in R. We use the rbeta() function to create 100 random beta samples with shape parameters 20 and 80. Then we plot this curve as the black line in Figure 8.2. Now suppose I take 15 more shots and make 5 of them. This will make a=30 and b=85. This updated curve is shown in red in Figure 8.2. The code below shows how to create the random beta samples with `rbeta(n, shape1, shape2)`. What will x look like? it is a vector of 100 random numbers drawn from a beta distribution with parameters a=20 and b=80. The mean will be 0.2 with the min around 0.1 and the max around 0.3. The code then draws the original curve in black and the updated curve in red. The `par(new=TRUE))` is used when you want to plot over an existing plot.

Code 8.4.2 — Basketball. A Beta Distribution.

```
x <- rbeta(100, 20, 80)
curve(dbeta(x, 20, 80), xlab=" ", ylab=" ", xlim=c(0.1,0.6),
     ylim=c(0,10))
par(new=TRUE)
curve(dbeta(x, 30, 85), xlab=" ", ylab=" ", xlim=c(0.1,0.6),
    ylim=c(0,10), col="red")
```

In the code and plot above, we updated the original black curve by adding baskets to a and misses to b. Our new probability given our data will be:

$$p(x = 1|D) = \frac{a+m}{a+b+m+l} \tag{8.15}$$

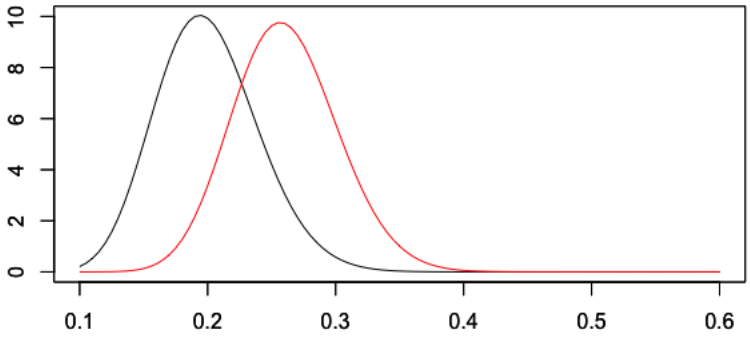


Figure 8.2: Beta Distribution for a=20, b=80

where m represents the number of new baskets and l represents the number of new losses.

The equation above is a Bayesian estimate. In contrast, note that the MLE estimate is simply m/N. As m and l approach infinity, the Bayesian estimate converges to the MLE. For a finite data set, the posterior probability with be between the prior and the MLE.

8.4.2 Multinomial and Dirichlet Distributions

We can extend the binomial distribution to the case where we have variables that are not binary but can take on more than 2 values. This is a multinomial distribution. The probability mass function of a multinomial distribution is:

$$Multinomial(m_1, m_2, ..., m_k | N, \mu) = \left(\frac{N}{m_1! m_2! ... m_k!}\right) \prod_{k=1}^{K} \mu_k^{m_k} \tag{8.16}$$

Where k indexes over the number of classes, K, and each of the m_i represent the probability of that class, with the sum of all $m_i = 1$.

The iris data is an example of a multinomial distribution. There are 3 classes, and the data set has 50 examples of each class, an even distribution. If we want to put the 150 flowers in 3 boxes (classes) with even probability of being in each class, we could use the following R command.

Code 8.4.3 — Multinomial. Iris Example

```
rmultinom(n=10, size=150, prob=c(1/3, 1/3, 1/3))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   56   44   51   55   54   47   49   60   61    42
[2,]   36   48   54   44   42   46   58   50   40    61
[3,]   58   58   45   51   54   57   43   40   49    47
```

The output above shows us 10 vectors, left to right, because we said "n=10". Our size is 150 for each of our iris flowers, and they have an even distribution. What we see in each column are distributions of the 150 flowers. Each column sums to 150. What we see in each row are the number of flowers in each box (class). The mean values for the 3 classes are 54, 50.1, and 45.9.

If we selected 6 flowers at random, what is the probability that there will be 1 flower from class 1, 2 flowers from class 2, and 3 flowers from class 3?

Code 8.4.4 — Multinomial. Use dmultinom() for probabilities.

```
dmultinom(x=c(1,2,3), prob=c(1/3, 1/3, 1/3))
[1] 0.08230453
# check:
factorial(6)/(factorial(3)*factorial(2)*factorial(1))*
    0.333333^1*0.333333^2*0.333333^3
[1] 0.08230403
```

The Dirichlet distribution is the conjugate prior of the multinomial distribution. The Dirichlet distribution has k parameters, α, one for each class. So instead of X being 0 or 1, it can take on k values. In the following, α_0 is the sum of all alphas. The Dirichlet distribution:

$$Dir(\mu|\alpha) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)...\Gamma(\alpha_k)} \prod_{k=1}^{K} \mu_k^{\alpha_{k-1}} \tag{8.17}$$

Consider a magic bag containing balls of K=3 colors: red, blue, yellow. For each of N draws, you place the ball back in the bag with an *additional* ball of the same color. As N approaches infinity, the colored balls in the magically expanded bag will be $Dir(\alpha_1, \alpha_2, \alpha_3)$.

You can think of the Dirichlet distribution as a multinomial factory.

Code 8.4.5 — Dirichlet. Output Distribution.

```
library(MCMCpack) # for function rdirichlet()
m <- rdirichlet(10, c(1, 1, 1))
m
              [,1]      [,2]       [,3]
 [1,] 0.015740801 0.3900641 0.59419507
 [2,] 0.295649733 0.3622780 0.34207224
 [3,] 0.464984547 0.4516325 0.08338300
 [4,] 0.365099590 0.3074731 0.32742729
 [5,] 0.065993901 0.2832624 0.65074371
. . .
 [9,] 0.005201826 0.3076536 0.68714457
[10,] 0.326711959 0.4160060 0.25728208
```

```
mean(m[,1])
```

```
[1] 0.2139159
> mean(m[,2])
[1] 0.3899679
> mean(m[,3])
[1] 0.3961162
```

We asked for 10 distributions with our alphas all equal to 1. The sum of every row, which is every distribution, is 1.0. The mean of the columns for these 10 examples are 0.2, 0.38, and 0.39. When run with 1000 examples the means were 0.34, 0.33 and 0.32.

8.4.3 Gaussian

The Gaussian or normal distribution is used for quantitative variables. Two parameters define its shape: the mean μ and the variance σ^2. The probability density function is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{8.18}$$

Let's plot a few Gaussians to see how the mean and variance influence the shape. All 3 distributions are generated with the dnorm() function, and all have a mean of 0. Different means would shift the curves right or left. The three curves have different standard deviations.

Code 8.4.6 — Gaussians. Normal distributions.

```
curve( dnorm(x,0,1), xlim=c(-4,4), ylim=c(0,1) )
curve( dnorm(x,0,2), add=T, col='blue' )
curve( dnorm(x,0,.5), add=T, col='orange' )
```

The rnorm() function in R generates random numbers, following a normal distribution. The d in the dnorm() function is for density, as in pdf, probability density function. The dnorm() function returns the pdf for the normal distribution specified by the parameters.

The pdf of the Gaussian above in Equation 5.18 is for a single variable x. The Gaussian can be extended to a D-dimensional vector **x** in which case it is called a multivariate Gaussian and has the pdf shown below where μ is now a vector of means, Σ is a DxD covariance matrix, $|\Sigma|$ is the determinant.

$$f(x) = \frac{1}{\sqrt[D/2]{2\pi}\sqrt{|\Sigma|}} exp\left(-\frac{(x-\mu)^T(x-\mu)}{2\Sigma}\right) \tag{8.19}$$

8.5 Likelihood versus Probability

The sum of probabilities for all outcomes of an experiment is 1. This is a binomial distribution. This block of code computes this manually.

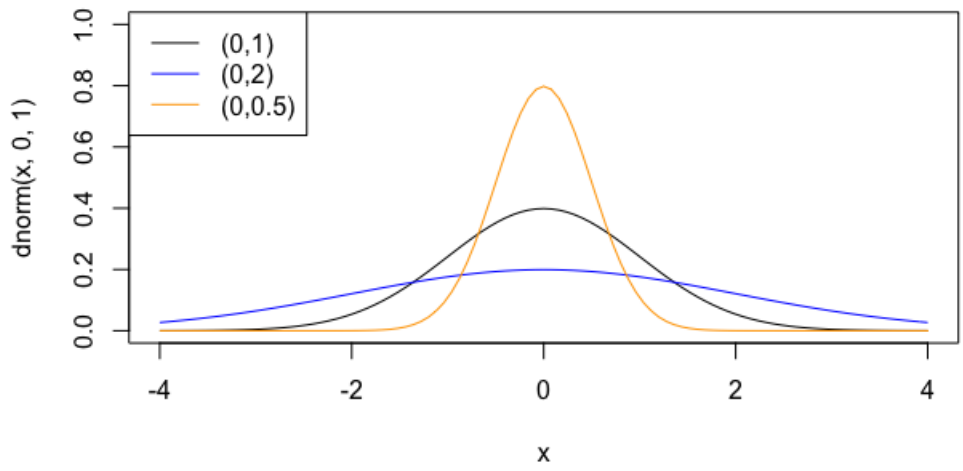


Figure 8.3: Gaussian Distributions

```
# 4 coin tosses, 5 possible outcomes

heads <- c(0, 1, 2, 3, 4)  # number of heads
tails <- c(4, 3, 2, 1, 0) # number of tails
binomial_coeff <- c(1, 4, 6, 4, 1)  # binomial coefficient
p_head <- 0.5
p_tail <- 0.5

# sum probability
p_vector <- rep(0, 5)  # 5 possible outcomes
for (i in 1:5){
  p_vector[i] <- binomial_coeff[i] * p_head^heads[i]  * p_tail^tails[i]
}
print(p_vector)  # prob for each possible event
[1] 0.0625 0.2500 0.3750 0.2500 0.0625
print(sum_prob) # sum of all probabilities
[1] 1
```

Now, let's try to find p_head given some coin tosses. Let's say for 4 coin tosses, there were 3 heads.

```
# out of 4 coin tosses, 3 were heads
# p = .75, the most likely value of p given the outcome
# compare this to p = .5 to see which is more likely
```

```
# look at the case where heads=3 tails=1
p <- 0.75
lh1 <- 6 * p^3 * (1 - p)^1
lh1
> lh1
[1] 0.6328125

p <- 0.5
lh2 <- 6 * p^3 * (1 - p)^1
lh2
[1] 0.375
```

The likelihood that p=0.75 is greater, given the data that we have seen.

Probability looks forward in time. We have p and can estimate outcomes based on p. Likelihood looks backward in time. We observe some data, and try to find the parameter p which most likely resulted in this outcome.

8.6 The Algorithm

Calculating joint probabilities with the chain rule above would be mathematically intractable. The simplifying assumption of the naive Bayes algorithm is that each of the predictors is independent. Therefore:

$$p(X_1, X_2, ...X_D|Y) = \prod_{i=1}^{D} p(X_i|Y) \tag{8.20}$$

The naive assumption of the independence of the predictors is typically not true, but perhaps surprisingly, naive Bayes works well. Naive Bayes forms a good baseline for comparing other classifiers.

The algorithm requires a single read through the data to estimate parameters. There are two sets of parameters to estimate from the data and two ways we can estimate the from the test data. The two methods are maximum likelihood estimates (MLE) and maximum apriori (MAP). MLE simply involves counting instances in the training data while MAP additionally makes some estimates based on prior distributions of the data. The two sets of parameters are counts for the probability of each class, and parameters for each predictor.

The first set of parameters to estimate is the probability of each class. If we estimate this with MLE, we just calculate the number of observations in each class. The estimate for class c will be the count of observations with class c divided by the number of observations:

$$MLE_c = \frac{|Y = y_c|}{|N|} \tag{8.21}$$

Estimating parameters for predictors depends upon their type. Binary features can use the mean of the Bernoulli distribution to get probabilities for each class. For discrete variables with more than 2 categories, the mean of the multinoulli distribution for each category can be used. Parameters for quantitative predictors are estimated from the mean and variance of the Gaussian distribution.

The MLE for the likelihood of a predictor given the class is also achieved by counting the data for discrete predictors. For predictor X_i and class c:

$$\hat{\theta}_{ic} = \frac{|X_{ic}|}{|N_c|} \tag{8.22}$$

It is possible that a given predictor for a given class may have 0 observations. In this case the estimate is 0 which is a problem given the multiplication of predictor likelihoods. An approach that eliminates this problem is smoothing, which involves adding a little to the numerator and denominator:

$$\hat{\theta}_{ic} = \frac{|X_{ic}| + l}{|N_c| + lm} \tag{8.23}$$

The value added to the denominator, m, represents the number of categories of X_i. If l=1 it is called Laplace or add-one smoothing. If we let l be a larger value this corresponds to a MAP estimate for the likelihood. We can add smoothing in a similar way to the MLE for the prior, in effect making it a MAP estimate.

For continuous variables, the mean and standard deviation of the predictor can be estimated but we would really like these values as they are associated with each class. Therefore separate mean, μ, and standard deviation, σ^2, values are computed by class. This is called a Gaussian naive Bayes classifier.

$$\hat{\theta_{ic}} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{8.24}$$

8.7 Applying the Bayes Theorem

There are a lot of misconceptions about probabilities. Let's walk through an example to illustrate some of the subtleties. Suppose that a given test for cancer has a sensitivity of 80% which means that if you have cancer, it will be positive with p=0.8. If your test is positive this does not mean you have an 80% chance of having cancer. Other data points we need: the probability of a false positive rate for the test is 10%, and the overall probability of having cancer is 0.4%. By plugging in the numbers into the theorem we say that the probability of actually having cancer given this positive test is 3.1%.

$$p(cancer|positive\_test) = \frac{0.8 * 0.004}{0.8 * 0004 + 0.1 * 0.996} = 0.031 \tag{8.25}$$

8.8 Handling Text Data

Previously all of our data has been numeric. However, machine learning with text is a hot topic in the field of Natural Language Processing. That's a separate course but we will talk a little about how text can be handled numerically now and revisit this in the data wrangling chapter. Any kind of natural language processing involves a lot of preprocessing. In a bag of words model, the text is tokenized, meaning that it is divided into individual words. The sentence structure is lost but just looking at counts of words can be informative. Additional preprocessing may involve removing punctuation, numbers, and making all text lowercase. Further, stop words are often removed. Stop words are common words that glue a sentence together but don't add much content. In this sentence, the stop words might be: *in, this, the, might, be*. Finally, the vocabulary size may be reduced since rare words don't have predictive value.

Often a document-term matrix is created in which each row represents a document and each column represents a term in the vocabulary. The intersection gives the count of a specific term in a given document. This is a sparse matrix in that each document has relatively few words out of the total vocabulary. Sometimes it is converted to a binary matrix with counts being replaced by 1 indicating the presense of that word in the document. The github site has a couple of examples of handling text data in the Data Wrangling section.

8.9 Naive Bayes v. Logistic Regression

Next we look at a classification data set in package mlbench. This package collects several real-world and artificial data sets for benchmarking. You can see the list of data sets by typing `data(package="mlbench")` at the console. We will use the BreastCancer data set, sometimes called the Wisconsin breast cancer data since the data originated from clinical practice at the Univeristy of Madison Wisconsin hospital. This data set has 669 observations with 11 columns. Column 1 is an ID that will be ignored, columns 2-10 are factors specifying information gleaned from biopsies. The final column is the label: benign or malignant.

8.9.1 Compare to Logistic Regression

Of the 669 observations in the breast cancer data set, approximately 64% are benign to 36% malignant. This is a reasonably balanced data set. We can easily use the summary() function to get the numbers: `summary(BreastCancer$Class)` gives us 458 benign and 241 malignant. Although breast cancer is rare in the general population, the data is fairly balanced, probably because patients going for a biopsy are likely to have suspicious lumps or other symptoms, and therefore be more likely to have cancer than the general population. In the notebook available online we first divide into 80% train and 20% test data, after removing the Id column. Then we perform logistic regression and get 91% accuracy on predicting benign or malignant on the test data. The accuracy of a classifier on test data can be further examined by breaking it down into sensitivity and specificity.

8.9.2 Naive Bayes Model

Next we build a Naive Bayes model on the same train and test data. The accuracy was 96%, higher than for the logistic regression model.

Code 8.9.1 — Naive Bayes. On the Breast Cancer Data.

```
nb1 <- naiveBayes(train[,-10], train[,10])
pred2 <- predict(nb1, newdata=test[,-10], type="class")
confusionMatrix(pred2, test$Class, positive="malignant")
```

8.9.3 Sparse data issues

Although both algorithms learned well from this data, digging into the training results shows some signs of difficulty. For example, there are 10 levels to the Cell.shape predictor, the Cell.size predictor and multiple levels for all other predictors. Let's say an average of 7 levels for each of 10 predictors, that's essentially 70 predictors. The training data has a little over 500 examples, that averages to about 7 examples per predictor level if they were distributed evenly, which they are not. This is a data sparseness problem. Ideally the number of examples should be much higher than the number of predictors. Fortunately, logistic regression and Naive Bayes are two algorithm that deal well with sparse data.

Looking in the online notebook at the summary for logistic regression, none of the 70+ predictors got a low p-value. Actually they all got p-values of 1.0! The summary shows that coefficients were assigned but that the standard error for each one is very large.

Looking at the results for the naive Bayes trained model, we see that the probabilities have to be spread out among the many levels. Look at the probabilities for Cell.size:

```
Cell.size
Y                    1           2           3           4
  benign    0.835616438 0.082191781 0.057534247 0.010958904
  malignant 0.010309278 0.036082474 0.113402062 0.139175258
            Cell.size
Y                    5           6           7           8
  benign    0.000000000 0.005479452 0.002739726 0.002739726
  malignant 0.118556701 0.103092784 0.082474227 0.113402062
            Cell.size
Y                    9          10
  benign    0.002739726 0.000000000
  malignant 0.015463918 0.268041237
```

Notice that nearly 84% of the probability for benign is taken by Cell.size 1, with diminishing values all the way to a zero probability for Cell.size 10. The malignant probabilities are more evenly distributed with the highest probability of 27% being for Cell.size 10.

8.9.4 Data preprocessing

Let's see how the two algorithms perform on a simplified version of the data set. THe data set was first subset to 3 columns: Class and two predictors: Cell.size and Cell.shape. Then the two predictors were converted to binary factors 0/1 where 1 indicates levels > 5. The code is shown below. These two predictors were chosen out of the 9 predictors randomly. Important predictors could have been omitted, and the arbitrary cut-off point at level 5 may not be optimal. All these types of decisions should be made with domain experts.

Code 8.9.2 — Data Preprocessing. Simplified train/test

```
df2 <- df[, c(2:3, 10)]   # just Cell.size Cell.shape and Class
df2$Cell.size <- as.factor(ifelse(df$Cell.size > 5, 1, 0))
df2$Cell.shape <- as.factor(ifelse(df$Cell.shape > 5, 1, 0))
str(df2)

train2 <- df2[i,]
test2 <- df2[-i,]
```

Running the logistic regression model on this data increased the accuracy from .91 to .92 but more significantly, the two predictors had p-values near 0. Running naive Bayes on the reduced data *decreased* the accuracy from .96 to .92. The naive Bayes algorithm was able to get the best model on all the data levels while logistic regression struggled to assign appropriate coefficients to each level of each predictor. The advantage to both models of the second simpler data set is that they are more interpretable.

One concern in looking at the conditional probabilities of the second naive Bayes model is that the two predictors separated well for the benign class but not so much for the malignant class. Notice that for the malignant class, it was almost as bad as tossing a coin. That's not what most of us would want for a diagnostic.

```
A-priori probabilities:
Y
   benign malignant
0.6529517 0.3470483

Conditional probabilities:
           Cell.size
Y                     0          1
  benign    0.98630137 0.01369863
  malignant 0.41752577 0.58247423

           Cell.shape
Y                     0          1
  benign    0.98630137 0.01369863
  malignant 0.40721649 0.59278351
```

8.9.5 Generative v. Discriminative Classifiers

In this example Naive Bayes outperformed logistic regression. It is important to keep in mind that these are two quite different classifiers. Logistic regression directly estimates the parameters of P(Y|X). This is called a *discriminative classifier*. Naive Bayes directly estimates parameters for P(Y) and P(X|Y). This is called a *generative classifier*. If the naive Bayes independence assumptions hold, and the number of training examples grows towards infinity, the naive Bayes and logistic regression converge toward similar classifiers. In general, Naive Bayes will do better with small data sets and logistic regression will do better as the size of the data grows. Naive Bayes has higher bias but lower variance than logistic regression. In this example, if you look at the output of the summary() function for the logistic regression model online you will see dozens of predictors because each of the 9 predictor columns are broken down into their factors. None of these almost 100 predictors achieved a low p-value and many factor levels were not included in the model. Further, five of the 9 predictor columns are ordinal factors which not only classify different values but there is an order to the values. The logistic regression function may have been overwhelmed by the sheer number of factors and levels whereas the simpicity of the Naive Bayes approach may have worked in its favor.

8.10 Naive Bayes from Scratch

In order to understand Naive Bayes on a deeper level, we will explore creating the algorithm from scratch. This will be applied to the Titanic data. The notebook online first loads and cleans the data, then runs the naiveBayes() function on the data.

8.10.1 Probability Tables

Here are the probability tables from the naiveBayes() function:

```
A-priori probabilities:
df[, 2]
       0        1
0.618029 0.381971
Conditional probabilities:
       pclass
df[, 2]         1         2         3
      0 0.1520396 0.1953028 0.6526576
      1 0.4000000 0.2380000 0.3620000
       sex
df[, 2]  female      male
      0 0.1569839 0.8430161
      1 0.6780000 0.3220000
       age
df[, 2]     [,1]     [,2]
      0 29.94757 12.22384
      1 28.78417 13.92003
```

Calculating the prior (apriori) probabilities of survived or perished is easy. It is simply dividing the counts of survived or perished by the total number of observations, as shown here:

```
apriori <- c(
  nrow(df[df$survived=="0",])/nrow(df),
  nrow(df[df$survived=="1",])/nrow(df)
  )
print("Prior probability, survived=no, survived=yes:")
[1] 0.618029 0.381971
```

8.10.2 Conditional Probability for Discrete Data

The conditional probability tables are also quite simple to create for qualitative data. Following Equation 7.21, we have a count for each level of each predictor.

```
# get survived counts for no and yes
count_survived <- c(
  length(df$survived[df$survived=="0"]),
  length(df$survived[df$survived=="1"])
)

# likelihood for pclass
lh_pclass <- matrix(rep(0,6), ncol=3)
for (sv in c("0", "1")){
  for (pc in c("1","2","3")) {
    lh_pclass[as.integer(sv)+1, as.integer(pc)] <-
      nrow(df[df$pclass==pc & df$survived==sv,]) /
       count_survived[as.integer(sv)+1]
  }
}          [,1]      [,2]      [,3]
[1,] 0.1520396 0.1953028 0.6526576
[2,] 0.4000000 0.2380000 0.3620000

# likelihood for sex
lh_sex <- matrix(rep(0,4), ncol=2)
for (sv in c("0", "1")){
  for (sx in c(2, 3)) {
    lh_sex[as.integer(sv)+1, sx-1] <-
      nrow(df[as.integer(df$sex)==sx & df$survived==sv,]) /
       count_survived[as.integer(sv)+1]
  }
}          [,1]      [,2]
[1,] 0.1569839 0.8430161
[2,] 0.6780000 0.3220000
```

8.10.3 Likelihood for Continuous Data

To calculate the likelihood for age we first need the mean and variance.

```
age_mean <- c(0, 0)
age_var <- c(0, 0)
for (sv in c("0", "1")){
    age_mean[as.integer(sv)+1] <-
      mean(df$age[df$survived==sv])
    age_var[as.integer(sv)+1] <-
      var(df$age[df$survived==sv])
}
age_mean
[1] 29.94757 28.78417
> age_var
[1] 149.4223 193.7673
```

Now we plug these values into Equation 7.23. We will write a function to calculate this, using the R built-in functions exp( ).

```
calc_age_lh <- function(v, mean_v, var_v){
  # run like this: calc_age_lh(6, 25.9, 138)
  1 / sqrt(2 * pi * var_v) * exp(-((v-mean_v)^2)/(2 * var_v))
}
```

8.10.4 Putting it All Together

Now we need a function to caculate Bayes' theorem for us.

```
calc_raw_prob <- function(pclass, sex, age) {
  # pclass=1,2,3  sex=1,2   age=numeric
  num_s <- lh_pclass[2, pclass] * lh_sex[2, sex] * apriori[2] *
      calc_age_lh(age, age_mean[2], age_var[2])
  num_p <- lh_pclass[1, pclass] * lh_sex[1, sex] * apriori[1] *
      calc_age_lh(age, age_mean[1], age_var[1])
  denominator <- lh_pclass[2, pclass]  * lh_sex[2, sex] *
   calc_age_lh(age, age_mean[2], age_var[2]) * apriori[2] +
      lh_pclass[1, pclass]  * lh_sex[1, sex] *
      calc_age_lh(age, age_mean[1], age_var[1]) * apriori[1]
  return (list(prob_survived <- num_s / denominator,
          prob_perished <- num_p / denominator))
}
}
```

Separately, we create a numerator for survived, and one for perished. The denominator multiplies the likelihood for pclass times sex times age for perishing times the prior probability of perishing, then does the same for surviving, adding these two values together.

Let's call this function for the first 5 test observations.

```
for (i in 1:5){
  raw <- calc_raw_prob(test[i,1], as.integer(test[i,3]), test[i,4])
  print(paste(raw[2], raw[1]))
}
```

8.10.5 Results

The following shows the predictions:

```
[1] "0.134219499226771 0.865780500773229"
[1] "0.119544295476936 0.880455704523064"
[1] "0.135715780701606 0.864284219298394"
[1] "0.267316737470339 0.732683262529661"
[1] "0.649768435768306 0.350231564231694"
```

Below are the raw probabilities from the Naive Bayes model. Notice they are the same.

```
> pred[1:5,]
             0         1
[1,] 0.1342195 0.8657805
[2,] 0.1195443 0.8804557
[3,] 0.1357158 0.8642842
[4,] 0.2673167 0.7326833
[5,] 0.6497684 0.3502316
```

The above code detailed how the algorithm works: counting and simple math. The code was not written in the most efficient way for R but in the way that makes how the algorithm works most clear to human readers.

8.11 Summary

Naive Bayes is a dependable classifier that is often used as a baseline for more sophisticated algorithms that are expected to outperform it. However, Naive Bayes is a reliable classifier in its own right. Its simple, probabilistic results make interpretation of the learning easy. Naive Bayes may outperform more sophisticated algorithms on small data sets. Next we look at the strengths and weaknesses of Naive Bayes.

Strengths:
- Works well with small data sets
- Easy to implement
- Easy to interpret
- Handles high dimensions well

Weaknesses:
- May be outperformed by other classifiers for larger data sets
- Guesses are made for values in the test set that did not occur in the training data
- If the predictors are not independent, the naive assumption that they are may limit the performance of the algorithm

8.11.1 New Terminology in this Chapter

This chapter used a lot of terminology from probability theory:
- likelihood v. probability
- prior probability v. posterior probability
- Bayes Theorem
- conditional probability
- joint probability
- marginal probability
- expected values, mean and variance

In addition we reviewed several probability distributions by family:
- Bernoulli, binomial and beta distributions
- Multinomial and Dirichlet distributions
- Gaussian distributions

Finally, there are a few terms related to techniques:
- MLE maximum likelihood estimate
- MAP maximum apriori estimate
- Laplace smoothing
- Discriminative v. generative classifiers

8.11.2 Quick Reference

Reference 8.11.1 Build a Naive Bayes Model

```
library(e1071)
# method one: use a formula
nb_model <- naiveBayes(formula, data=train)
```

Reference 8.11.2 Build a Naive Bayes Model

```
# method two: X, Y
nb_model <- naiveBayes(predictor_cols, target_col, data=train
```

Reference 8.11.3 Predict

```
raw <- predict(model, newdata=test, type="raw")
pred <- predict(model, newdata=test, type="class"
```

Reference 8.11.4 Confusion Matrix

```
library(caret)
confusionMatrix(predictions, test$target, positive="2")
```

8.11.3 Practice to Consolidate Skills

Problem 8.1 — Classification on the Abalone Data. Try the following:

- Re-run your code from the Chapter 5 Lab on classifying the Abalone data using Logistic regression, or follow those instructions to create the model.
- Create a naive Bayes model on the same train/test split.
- Compare the performance of the two algorithms.
- Compare the confusion matrix tables of the two algorithms. What do you observe? Was one better at predicting true positives versus true negatives? Why might this be important?

Problem 8.2 — Classification on the Heart Data. Try the following:

- Re-run your code from the Chapter 5 Lab on classifying the Heart data using Logistic regression, or follow those instructions to create the model.
- Create a naive Bayes model on the same train/test split.
- Compare the performance of the two algorithms.
- Compare the confusion matrix tables of the two algorithms. What do you observe? Was one better at predicting true positives versus true negatives? Why might this be important? How many NAs were predicted for each algorithm?

Problem 8.3 — Classification on the Sonar Data. Try the following:

- Load the Sonar data set from package mlbench. Research this data set and write a brief description of the columns.
- Divide the data into 80-20 train-test.
- Create a logistic regression model of the data. What is the accuracy?
- Create a naive Bayes model of the data. Compare the accuracy to the logistic regression model.
- Compare the confusion matrix tables for each. Discuss what you find.

Problem 8.4 Compare how logistic regression makes classification predictions compared to naive Bayes.

Problem 8.5 Briefly summarize why logistic regression is called a discriminative classifier and naive Bayes is called a generative classifier.

Problem 8.6 What is the naive assumption in Naive Bayes?

8.11.4 Next-Level Learning

Tom Mitchell's classic book, *Machine Learning* Chapter 3 discusses Naive Bayes and Logistic Regression. Tom Mitchell has provided free access to this chapter here:

http://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf

A well-regarded paper comparing discriminative and generative classifiers by Andrew Ng and Michael Jordan can be found here:

https://ai.stanford.edu/~ang/papers/nips01-discriminativegenerative.pdf

9. The Craft 2: Inductive Learning

Machine-learning expert Pedro Domingos summarizes machine learning as follows:

```
Learning = Representation + Evaluation + Optimization
```

Representation, evaluation, and optimization are the major tasks of machine learning. We have done all of them in the previous chapters, although we never explicitly said so. The algorithms we have used in R rely on internal metrics to *evaluate* how well the training is proceeding. Internal *optimization* code ensures that the best model is produced, given the training data.

In machine learning, we need to figure out two kinds of representation. First, we need to be able to state the problem in a formal language, so that computation can take place. Second, we need to decide how the input data will be represented.

For the algorithms learned so far, the input data was represented as (**x**, y) pairs of data, where **x** is a matrix of features, and y is the target value. The learning problem for linear classifiers is represented as:

$$y = \mathbf{w}\mathbf{X} + b \tag{9.1}$$

where **w** represents the weight matrix which is multiplied by the **X** input features plus the fitting parameter b. The parameters w and b are learned from the data.

9.1 How learning happens

Figure 9.1 gives a big-picture view of learning for optimization algorithms such as logistic regression and neural networks. Given a problem specification, an initial hypothesis can be

created, h_0. The first hypothesis is not likely to be the best, it may just be a random guess in some algorithms. The results of this first hypothesis are evaluated, using an error function, E, which measures the distance between the predicted value and the actual value for y. This error is used to optimize the next hypothesis, h_1, which should be an improvement over the initial hypothesis. This process continues until a hypothesis is found that fits the training data to some acceptable threshold level.

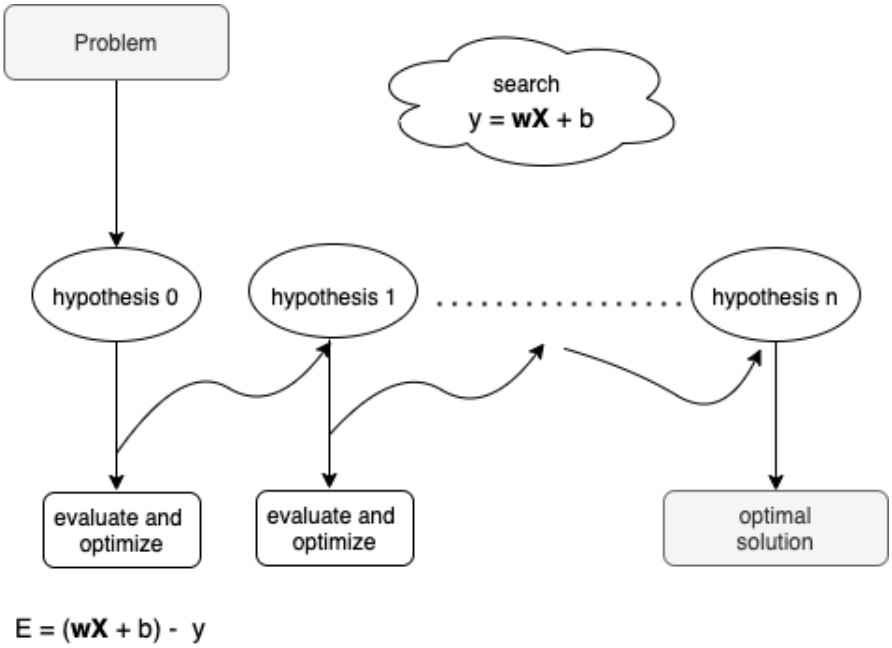


Figure 9.1: Learning as Search

Each hypothesis for a linear model, is a matrix of values **w** that is learned from the data. We can think of machine learning as a search through the hypothesis space for the optimal hypothesis.

This form of machine learning is *inductive learning*. The algorithm is given examples of training data, and has to learn a hypothesis (or function) that models the data. Once this model or function is learned, it can be used to make predictions on previously unseen data that is from a similar distribution as the training data. The extent to which a model can generalize to new data, the better it will perform. Unfortunately, machine learning algorithms can have some degree of *inductive bias*, the assumptions that the model makes about the incoming data.

A recent article from MIT Technology Review[1] observed that real-world performance on machine learning models is often disappointing compared to results in the lab because the real-world data may be different. This is the *data shift* problem. Another reason for disappointing real-world results is *underspecification*, a term borrowed from statistics where it indicates that some missing or hidden variables are unaccounted for in the model. In machine

[1] `https://www.technologyreview.com/2020/11/18/1012234/`

learning, underspecification can also result from the way that some algorithms learning by starting from random parameter settings. One proposed solution is to move away from the train-test-deploy paradigm. Instead, multiple models would be created that perform well on in-lab testing data. Then, these models would be deployed to operate on real-world data. A period of real-world evaluation could identify the best model.

9.2 Feature Selection

Another problem in machine learning is determining whether to use all available features, or to select a subset of them. Many of the algorithms we have used have embedded methods to determine the usefulness of predictors. For example, linear and logistic regression give us p-values so that we can gauge confidence in the predictive value of variables. Others do not so we need to use external methods.

9.2.1 Feature Selection with caret

The caret package has several useful functions for feature selection, we will explore a couple of them here. First, the findCorrelation() function which returns a list of columns that can be removed due to their correlation with other variables. We will use the PimaIndiansDiabetes2 data from package mlbench. First we remove rows that have NAs, then compute a correlation matrix with cor() and input this into the findCorrelation() function. The function lets you specify the cutoff point. In this case we had a cutoff of .5 correlation. We also set verbose to TRUE to get fuller information from the function.

Code 9.2.1 — Find Correlated Variables. PimaIndiansDiabetes2.

```
library(caret)
library(mlbench)
data("PimaIndiansDiabetes2")
df <- PimaIndiansDiabetes2[complete.cases(PimaIndiansDiabetes2[]),]
corMatrix <- cor(df[,1:7])
findCorrelation(corMatrix, cutoff=0.5, verbose=TRUE)
```

```
Compare row 6  and column  4 with corr  0.664
  Means:  0.265 vs 0.187 so flagging column 6
Compare row 2  and column  5 with corr  0.581
  Means:  0.266 vs 0.161 so flagging column 2
All correlations <= 0.5
[1] 6 2
```

The output of findCorrelation() recommended that we remove column 6, mass, because it correlates with triceps. It also recommended that we remove column 2, glucose, because it correlates with insulin. This is easily done with: `df <- df[,-c(2,6)]` Recall that in this

data set we had a lot of NAs for triceps. Therfore, knowing that mass correlates with triceps but mass has very few NAs might cause us to remove triceps instead of mass.

9.2.2 Ranking Feature Importance

Of the variables we have left, which ones are most important? The caret package was used in the code below to train a model and extract variable importance. We show the output below the code and the plot in Figure 9.2.

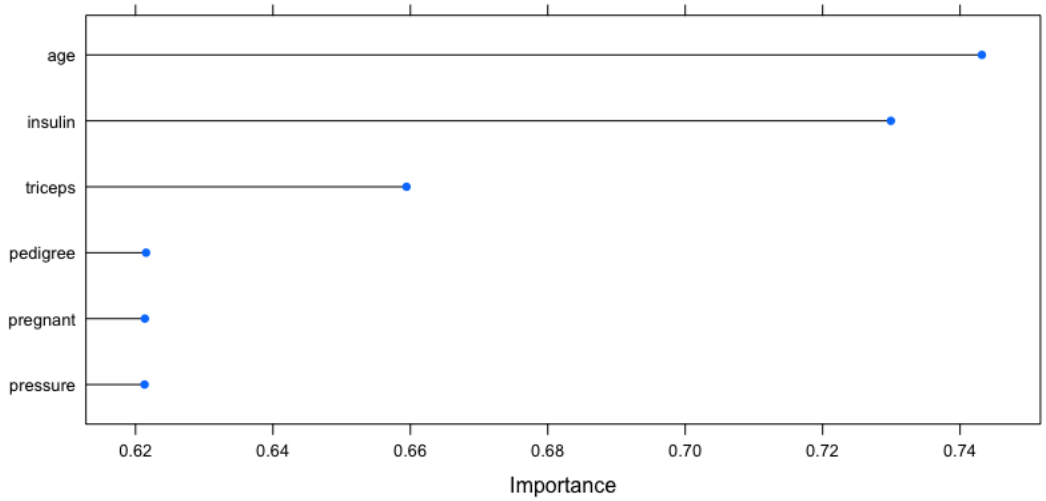


Figure 9.2: Plot of Variable Importance

Code 9.2.2 — Ranking Feature Importance. PimaIndiansDiabetes2.

```
ctrl <- trainControl(method="repeatedcv", repeats=5)
model <- train(diabetes~., data=df, method="knn",
     preProcess="scale", trControl=ctrl)
importance <- varImp(model, scale=FALSE)
importance
plot(importance)
```

```
ROC curve variable importance

         Importance
age          0.7432
insulin      0.7299
triceps      0.6594
pedigree     0.6215
pregnant     0.6214
pressure     0.6213
```

9.2.3 Recursive Feature Selection

Another option in caret is recursive feature selection. This recursively eleminates features to find a subset of predictors that perform well. We start will all the original features. This gave us different advice than the findCorrelation() function above. Here we are told we can eliminate mass and pedigree but it left in both glucose and insulin which we know are highly correlated.

Code 9.2.3 — Recursive Feature Selection. PimaIndiansDiabetes2.

```
df <- PimaIndiansDiabetes2[complete.cases(PimaIndiansDiabetes2[]),]
ctrl <- rfeControl(functions=rfFuncs, method="cv", number=10)
rfe_out <- rfe(df[,1:7], df[,8], sizes=c(1:7), rfeControl=ctrl)
rfe_out
```

```
Recursive feature selection
Outer resampling method: Cross-Validated (10 fold)
Resampling performance over subset size:
 Variables  RMSE Rsquared   MAE RMSESD RsquaredSD  MAESD Selected
         1 7.177   0.5144 5.176  1.768     0.1737 1.0658
         2 7.602   0.4676 5.433  1.738     0.1501 0.9455
         3 7.358   0.4916 5.350  1.754     0.1532 0.9457
         4 7.363   0.4949 5.372  1.760     0.1562 1.0373
         5 7.416   0.4909 5.432  1.650     0.1367 0.9637
         6 7.221   0.5120 5.276  1.658     0.1413 1.0005
         7 6.999   0.5459 5.126  1.725     0.1463 0.9692        *
The top 5 variables (out of 7):
   pregnant, glucose, insulin, triceps, pressure
```

9.3 FSelector

Another package that can help with feature selection is FSelector. Here is a sample run :

Code 9.3.1 — FSelector. PimaIndiansDiabetes2.

```
library(FSelector)
var_scores <- random.forest.importance(diabetes~., df)
```

```
pregnant 12.41986326
glucose 50.53806586
pressure 0.03753078
triceps 8.39019322
insulin 18.00908543
mass 14.08434636
pedigree 7.30103768
age 24.80577317
```

The best predictor was glucose, followed by age. Recall that caret recommended removing glucose because it correlated highly with insulin. As with any diagnosis, a second opinion never hurts.

9.4 Predictive Modeling

The previous chapters described regression and classification algorithms that can be used to model data, and then predict results on future unseen data. The approach described so far in the book has a good deal of overlap with the field of predictive modeling. A definition from the Gartner Glossary:[2]

> **Predictive modeling** is a commonly used statistical technique to predict future behavior. Predictive modeling solutions are a form of data-mining technology that works by analyzing historical and current data and generating a model to help predict future outcomes. In predictive modeling, data is collected, a statistical model is formulated, predictions are made, and the model is validated (or revised) as additional data becomes available.

The examples in this book are one-and-done small projects to illustrate a concept or technique. In a real-world predictive modeling project, the analysis continues in a cyclic manner, ever-evaluating and improving the model. Predictive modeling involves many techniques covered in this book: data preparation, model building, model evaluation, model improvement. Predictive modeling often starts with an objective. For example, a mobile phone service company may want to understand their customers who churn in order to predict customers that are likely to switch to another carrier. Notice that this model would not necessarily be useful for another carrier because the dynamics that make customers leave can be quite different between companies. The objective is typically defined by organization managers who may have little technical knowledge or experience with machine learning. It is important to set realistic expectations with management about the level of accuracy that can be built from the data at hand. Improving accuracy increases in difficulty the higher you go. Going from 98 to 99% accuracy may require much more data and more complex models. The more input features that are used, the more data is needed in order to let the algorithm see all possible values of all features.

Predictive modeling focuses on accuracy in prediction over other considerations such as interpretability. The three algorithms covered in the last three chapters are all highly interpretable. In linear regression, for example, the coefficients quantify the change in y for a one-unit change in x. Other algorithms covered later in the book are not very interpretable, but are more like black boxes.

A definitive guide to predictive modeling is *Applied Predictive Modeling* by Max Kuhn and Kjell Johnson. Max Kuhn is also the author of several R packages, including the caret package.

[2]https://www.gartner.com/en/information-technology/glossary/predictive-modeling)

Part III: Searching for Similarity

Preface to Part Three

Part Three explores machine learning algorithms that group observations by similarity. These algorithms are quite diverse, some using supervised and other unsupervised approaches. Some of the supervised techniques can be used for regression or classification. Some are often used for data analysis. These algorithms will form an important part of your skill set in learning from data.

- Chapter 10 looks at the kNN, k Nearest Neighbor algorithm, which can be used for regression or classification
- Chapter 11 looks at two different clustering algorithms: k-means and hierarchical clustering, both are unsupervised methods
- Chapter 12 explores decision trees, which can be used for regression or classification

10. Instance-Based Learning with kNN

10.1 Overview

The kNN algorithm is a supervised learning algorithm but it does not form a model of the input data. Instead, all the training observations are simply stored in memory. When a new observation needs to be evaluated, the algorithm compares it with the observations stored in memory, finding the closest k neighbors.

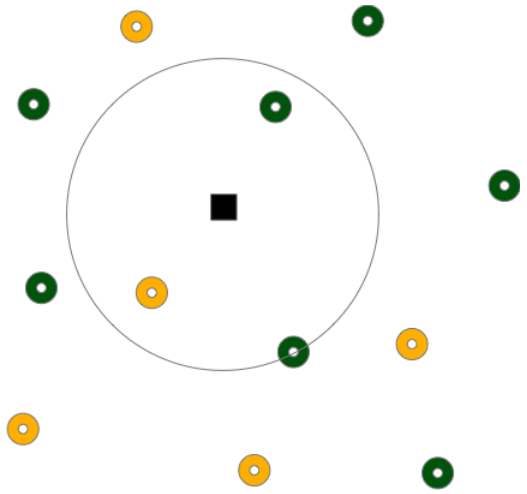


Figure 10.1: Finding k=3 Nearest Neighbors

As illustrated in Figure 10.1, the new observation is the black square. The three nearest neighbors were found and the observation will be classified as green because the majority

of near neighbors were green. If this had been a regression task, the black square would be predicted to have a value that is the average target value of the nearest neighbors.

10.2 kNN in R

Using the familiar iris data set we will run the kNN algorithm. Notice in Figure 10.2 that we have 3 classes. One of the nice things about the kNN algorithm is that it can predict class membership in a multi-class data set.

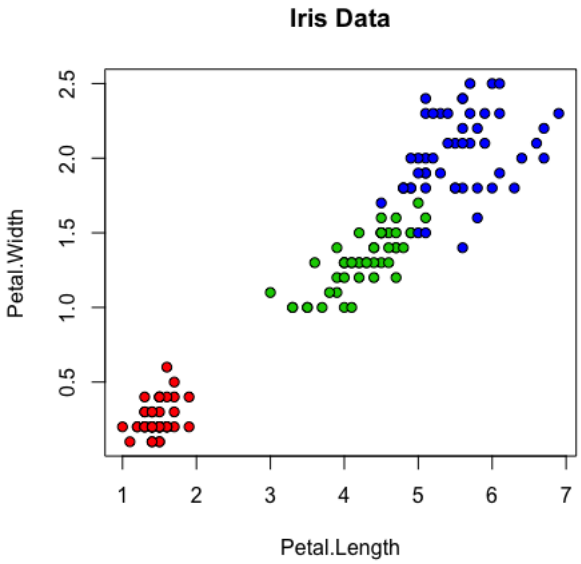


Figure 10.2: The Three Iris Classes

The code for running the knn algorithm on the iris data is shown below. In the notebook online, we have first randomly divided the 150 observations into 67% training and 33% test. We have also separated out the labels in both train and test into new vectors and removed the labels from train and test. There are 4 arguments to the knn() function below: the training data, the test data, the training labels and the chosen value of k. Notice that we do not build a model, we are loading into memory and predicting on the test data all in one command.

Code 10.2.1 — kNN Classification. The Iris Data.

```
library(class)
iris_pred <- knn(train=iris.train, test=iris.test,
     cl=iris.trainLabels, k=3)
```

After running the above code, the iris_pred variable will be a vector of class labels: setosa, virginica or versicolor. An optional parameter for knn() is to set prob=TRUE which will return probabilities rather than class predictions. The algorithm achieved 98% prediction accuracy

on the test data, but as we have seen before, and as you can observe in Figure 10.2, this is an easy data set to classify. The notebook code to compute accuracy was:

```
acc <- length(which(iris_pred == iris.testLabels)) /
       length(iris_pred)
```

R code can sometimes seem like those Russian nesting dolls so let's unpack from the inner function out. The `which(`*iris_pred == iris.testLabels*`)` returns a vector of indices for test items that were correct. The length() function surrounding this returned 49. This 49 was divided by the length of the predictions, 50, to get the 98% accuracy. The online notebook compares this result to performing one-versus-all classification with logistic regression.

Exercise 10.1 — kNN Classification on Wine Data. You can find the data set wine_all.csv in the github site. It is a 6497x13 data set of the chemical composition of red and white wines. This data set was edited from the wine data sets on the UCI ML Repository. Your task is to use kNN to classify red/white wine.

- Divide the data into train and test sets, setting a seed first for reproducibility.
- Use R commands to make sure that the train and test sets have distributions of white and red types similar to the overall data.
- For comparison, first build a logistic regression model predicting wine type (red or white) based on all other columns. What is your acccuracy?
- Run knn() with k=3 on the data and compare the accuracies of the two algorithms.

10.3 The Algorithm

In kNN learning, an observation is known by the company it keeps. For a given test observation x_i, the kNN classifier will identify the k closest points, the neighbors, and estimate the conditional probability for class j as the fraction of neighbors that have that class.

$$P(Y = j|X) = \frac{1}{k}\sum_i I(y_i = j) \tag{10.1}$$

where I() is an indicator function returning TRUE or FALSE.

For regression, an average of the neighbors' target value is taken to be the predicted value for an instance.

$$\hat{y} = \frac{1}{k}\sum_{i \in NB} y_i \tag{10.2}$$

where NB is the set of neighbors.

10.3.1 Choosing K

The choice of a value for k needs to be done before the algorithm is run. The choice of k is critical to the bias-variance tradeoff of the algorithm. If k is very small, the classifier will have low bias but high variance. As k grows, the algorithm becomes less flexible and bias increases while variance decreases. The optimal value for k is often found by cross validation.

A rule-of-thumb that is sometimes stated says to choose a k that is the square root of the number of observations. We have not noticed this to be an effective heuristic. Cross validation is much more reliable. If you are doing classification, it makes sense to let k be an odd number. If k is too small the algorithm will be susceptible to noise but if k is too large the computation time increases.

10.3.2 Curse of Dimensionality

The kNN algorithm works best when we have few predictors. If we have 3 predictors, it will be easier to find neighbors in this 3-dimensional space. If we have 20 predictors, it will be harder to find neighbors in this 20-dimensional space. This is referred to as the curse of dimensionality. Some algorithms, like Naive Bayes, do not suffer in high dimensions but kNN will bog down.

10.4 Mathematical Foundations

What does it mean for an instance to be near another? Often Euclidean distance is used:

$$dist = \sqrt{\sum_i (q_i - p_i)^2} \tag{10.3}$$

However, simple variance is more computationally efficient. In computing the distance or variance, the predictors need to be numeric. If a column is a factor, it should be either be ignored or converted to an integer for kNN.

10.4.1 Scaling Data

The terms scaling and normalization are used inconsistently in the literature. Generally, *normalization* means applying transformations to the data so that it follows a normal distribution while *scaling* implies some linear transformation of the data that may or may not result in a normal distribution. The R scale() function, using the default settings, will transform data to have a mean of 0 and a standard deviation of 1. If the data is normally distributed this should be sufficient. You can easily look at the distribution with a hist() graph. If data is highly skewed then you might get better results with something like this:

```
normalize <- function(x) {
  return((x - min(x)) / (max(x) - min(x)))
 }
 # apply to all columns
 data_norm <- as.data.frame(lapply(df, normalize))
```

Another nice thing about the R scale() function is that there is an unscale() function to convert scaled data back to the original by doing the inverse operations that scaled it. The unscale() function uses information stored with the scaled data.

```
library(DMwR)
scaled <- scale(df)
predictions <- predict(...)
original <- unscale(predictions, scaled)
```

In the sample code above we scaled all the data, including the targets, before dividing into train and test. Another approach is to divide the data, then scale train and test separately. When doing this, many experts advise to do scaling on both train and test using the mean and standard deviation of the train set. Why? Some experts feel that allowing the test data to be scaled by information from the entire data set is leaking information about the data to the test set, which is prohibited. Here is an approach that follows this advice:

```
# normalize data
means <- sapply(train, mean)
stdvs <- sapply(train, sd)
train <- scale(train, center=means, scale=stdvs)
test <- scale(test, center=means, scale=stdvs)
```

Using the approach above, train and test contain just the predictor columns. There is no need to scale the targets, which are held in other vectors. It is only the predictors that need to be scaled.

In the knn classification problem above on the iris data, we did not need to scale the predictors because all measurements are in the same units. As we will see in the next example, when predictors are not in the same units, scaling usually improves performance.

Exercise 10.2 — Scaling Data. Using the same wine data set as above:

- Scale the data and see if the knn performance improves.
- See if reducing the number of predictors (reducing the curse of dimensionality) improves the results of the knn algorithm.

▪

10.5 kNN Regression

Next we perform regression on the Auto data set in package ISLR. In the online notebook, first we do linear regression to predict mpg, miles per gallon, based on weight, year, and origin. The linear regression model gets a correlation of 0.89 and an mse of 14.6. The rmse will be around 3.8 mpg. Let's see if kNN can do better.

The first kNN attempt in the online notebook did not scale the data and got worse results than linear regression: correlation of 0.88 and mse of 17. Then we scaled the data. Running kNN on the scaled data resulted in a correlation of 0.91, which is better than the linear model, and an mse of 14. This is the best of the 3 models, the rmse tells us we are off an average of 3.8 mpg. Note that we use the knnreg() function in package caret.

Code 10.5.1 — kNN Regression. Auto Data.

```
library(caret)
fit <- knnreg(train[,2:4],train[,1],k=3)
predictions <- predict(fit, test[,2:4])
cor(predictions, test$mpg)
mse <- mean((predictions - test$mpg)^2)
```

Exercise 10.3 — kNN Regression. Using the same wine data set as above, we now try to predict the quality.

- Remove the red/wine column from the scaled train and test sets.
- Run knnreg() on the data.
- What is the cor() and mse?
- Create a linear regression model.
- Compare linear regression and knn model performance on the test set.

■

10.6 Find the Best K

The results with k=3 were good on the Auto data, but how do we know if a different k would produce even better results? We can try various values for k to find out. First we fill two vectors with 20 zeroes. These will hold our results for each level of k. Then we let k be 1, 3, 5, ..., 39. At each iteration in the for loop we store the correlation and mse and also print them out. Trying 1 to 39, skipping by 2, is purely arbitrary. You might try a larger range or skipping more or fewer possible k values.

Code 10.6.1 — Find the Best K. Auto Data

```
cor_k <- rep(0, 20)
mse_k <- rep(0, 20)
i <- 1
for (k in seq(1, 39, 2)){
  fit_k <- knnreg(train[,2:4],train[,1], k=k)
  pred_k <- predict(fit_k, test[,2:4])
  cor_k[i] <- cor(pred_k, test$mpg)
  mse_k[i] <- mean((pred_k - test$mpg)^2)
  print(paste("k=", k, cor_k[i], mse_k[i]))
  i <- i + 1
}
```

We can either visually go down the printed values to find the minimum mse and the maximum correlation or we can use the following commands at the console:

```
> which.min(mse_k)
```

```
[1] 8
> which.max(cor_k)
[1] 8
```

It looks like the 8th element is best. That corresponds to k=15. Let's plot this to get a visual understanding. The plot in Figure 10.3 confirms that k=15 (the 8th run) gives the best results. The MSE is in blue, we want that to be as low as possible. The correlation is in red, we want that to be as high as possible. The code below generated this plot. Notice the command `par(new=TRUE)`. This causes the next plot() to be added to the existing plot.

Code 10.6.2 — Plot mse and cor. For various k.

```
plot(1:20, cor_k, lwd=2, col='red')
par(new=TRUE)
plot(1:20, mse_k,lwd=2,col='blue',labels=FALSE,ylab="",yaxt='n')
```

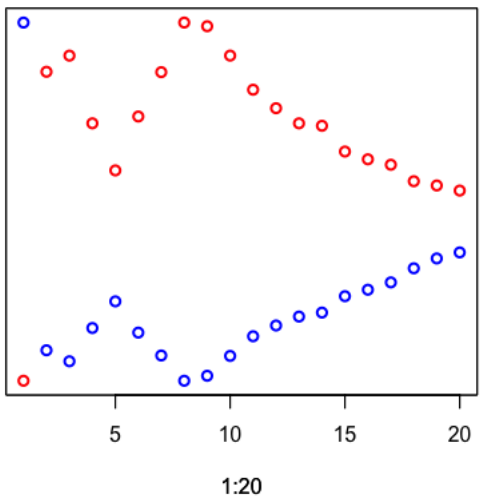


Figure 10.3: MSE (blue) and Correlation (red) for Various K

Exercise 10.4 — Finding the Best k. Using the same regression task as above, predicting quality from other variables, try the following:

- Find the best k for your model.
- What is the cor() and mse for this model? How does it compare to the linear regression model?

10.7 k-fold Cross Validation

The code sample in the last section ran through several values of k to find the optimal k. Running the model many times gave us more information about the data that running only once with k=3 would have. Techniques where algorithms are run different times to find the best or the most accurate parameters fall in the category of resampling methods. In effect the kNN algorithm is "sampling" neighbors. As seen in Figure 10.3 above, the results vary with the value of k. This shows how kNN is sensitive to the data. If we sampled the data many times to get different test sets, the results would vary quite a bit from sample to sample. There are many statistical approaches to sampling. In this section we will talk about k-fold cross validation.

10.7.1 Creating k Folds

Figure 10.4 illustrates 10-fold cross validation. In k-fold cross validation, the entire data set is divided into k (10 in this case) equal portions. For k iterations, the algorithm is trained on all but one portion of the data, leaving the held-out data for test. The test metrics for the k runs are then averaged together to get an overall estimate of test error. Common values of k are 5 and 10. Don't confuse k-NN with k-fold cross validation. The k values don't mean the same thing, it is simply an unfortunate coincidence that both use k.

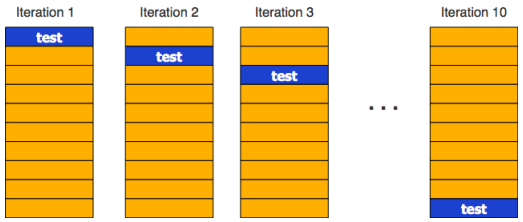


Figure 10.4: 10-fold Cross Validation

Since k-fold cross validation is computationally expensive, why do it? There are a few situations where it will be useful. One is if you have a very small data set and your algorithm is prone to high variance. By running the algorithm many times on different subsets of test data you will get a better idea of how the algorithm will generalize to new data. With just one sampling of the test data you may have gotten lucky or unlucky with the selection of observations in the test set.

Another reason to use cross validation is to select algorithm parameters. In the case of kNN we have to select a value for k. In the last section we selected this value based on how well it performed on this test set. Will that value of k also be good for other random draws of test data? Cross validation can answer that question. This is sometimes called *parameter tuning*.

Cross validation is a technique that can be used with any algorithm. For example, we could use it in linear regression by trying different polynomial regression lines to find the model that best fits the data. This is sometimes called *model selection*.

Next we use 10-fold cross validation on the knn regression problem we discussed earlier. This is in a kNN notebook in the github. After loading the Auto data and subsetting it to just

mpg, weight, year, and origin, we need to divide the data into folds. We could easily write the code ourselves but the `caret` package already has a nice function for that, `createFolds`. The first argument to createFolds() is the target column and the second tells it we want 10 folds. Since there are 392 observations in Auto, we expect a little less than 40 indices in each fold. We confirm that with sapply().

Code 10.7.1 — Cross Validation. Divide Data into Folds.

```
library(caret)
set.seed(1234)
folds <- createFolds(df$mpg, k=10)
sapply(folds, length)
Fold01 Fold02 Fold03 Fold04 Fold05 Fold06 Fold07 . . . Fold10
    39     41     37     40     39     39     39 . . .     39
```

You can look at the indices in the first fold with `folds[[i]]`. Recall that double square brackets are used with lists.

```
> folds[[3]]
 [1]   13   15   24   42   54   55   57   60   73   76   84   91 113 126 131 138
       143 144 157 159
[21] 164 173 191 197 204 206 207 218 225 230 270 278 298 310 339 351
       362 366
```

10.7.2 Run knnreg() on each Fold

Now that we have the folds, we can run knnreg() on each fold and average the results. For now we just let k=3.

Code 10.7.2 — Cross Validation. Run 10 times.

```
test_mse <- rep(0, 10)
test_cor <- rep(0, 10)
for (i in 1:10){
  fit <- knnreg(df[-folds[[i]], 2:4], df$mpg[-folds[[i]]], k=3)
  pred <- predict(fit, df[folds[[i]], 2:4])
  test_cor[i] <- cor(pred, df$mpg[folds[[i]]])
  test_mse[i] <- mean((pred - df$mpg[folds[[i]]])^2)
}
print(paste("Average correlation is ", round(mean(test_cor), 2)))
print(paste("range is ", range(test_cor)))
print(paste("Average mse is ", round(mean(test_mse), 2)))
print(paste("range is ", range(test_mse)))
```

```
[1] "Average correlation is  0.93"
[1] "range is  0.90883537818507"  "range is  0.946753085315825"
```

```
[1] "Average mse is  0.15"
[1] "range is  0.111756182643287" "range is  0.201818162667268"
```

You can see that the average correlation and mse are good and also that there is a wide range in both vectors. And this is with holding k steady at 3. This is a confirmation that kNN has high variance with a low value of k. Using the technique of running cross validation and averaging the results gives us a more complete picture of kNN performance on this data. Let's explore results if we use cross validation and vary k at the same time. This will involve rewriting the code.

10.7.3 Cross Validation with Various K

In the code section below we use sapply() to try different values of k on an anonymous function that does the 10-fold cross validation similarly to the previous code segment. The output of sapply() is stored in variable results. This will be a list consisting of the 40 output values, alternating the correlation and mse values. We'd like those separated so we coerce the list into a matrix m. Then column 1 of m will contain all the correlations and column 2 will contain all the mse values. Finally, there is code to create plots, which are shown in Figure 10.5. The graphs in the figure indicate that k=3 gives the highest correlation and the lowest mse.

Code 10.7.3 — Cross Validation. For k=1,3,5...

```
# try various values for k
k_values <- seq(1, 39, 2)
results <- sapply(k_values, function(k){
  mse_k <- rep(0, 10)
  cor_k <- rep(0, 10)
  for (i in 1:10){
    fit <- knnreg(df[-folds[[i]], 2:4], df$mpg[-folds[[i]]], k=k)
    pred <- predict(fit, df[folds[[i]], 2:4])
    cor_k[i] <- cor(pred, df$mpg[folds[[i]]])
    mse_k[i] <- mean((pred - df$mpg[folds[[i]]])^2)
  }
  list(mean(cor_k), mean(mse_k))
})
# reshape results into matrix
m <- matrix(results, nrow=20, ncol=2, byrow=TRUE)
```

Code 10.7.4 — Cross Validation. Plot Results

```
par(mfrow=c(2, 1))
plot(1:20, unlist(m[,1]), lwd=2, type="o", col='red',
     ylab="Correlation")
plot(1:20, unlist(m[,2]), lwd=2, type="o", col='blue', ylab="MSE")
```

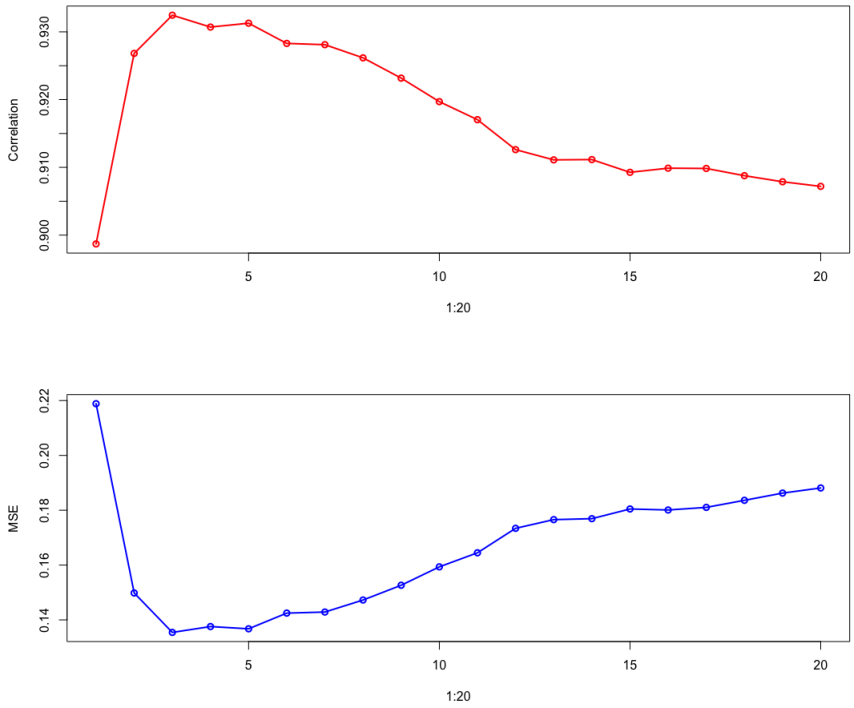


Figure 10.5: Cross Fold Validation for various K

10.7.4 Applying CV to Other Algorithms

The cross validation technique demonstrated in this section can be applied to any algorithm. However, this approach becomes significantly more time intensive with larger data. For example, if you have an algorithm on a large data set that takes 30 minutes to run, and you do 10-fold cross-validation, that will take 5 hours. The catch-22 with small data is that your small test set is likely to result in variance and yet if you put more of your limited data into the test set then you have less for training which could make your algorithm perform poorly. Cross-validation enabled us to have more confidence in our test set metrics by compensating for a relatively small data set. Researchers generally recommend values of k=5 or k=10 since these have been demonstrated empirically to create a good balance between bias and variance.

The `caret` package contains many functions related to sampling. This section had a regression example. If we are performing classification, we need to be concerned about the distribution of observations in the train and test sets. The downSample() function will randomly sample each class to be the size of the smallest class. This results in smaller data which is fine if you have a lot of data to start with. The upSample() function will randomly sample with replacement so that the smaller class becomes as large as the majority class. These are called *subsampling* techniques, and are described in the caret documentation.

10.8 Summary

The kNN algorithm is an example of an instance-based approach. It is often included in lists of clustering algorithms, although it doesn't cluster all the data, it just creates a cluster of neighbors around a test instance. The kNN algorithm is also sometimes called lazy learning because it doesn't do much until test time.

The kNN algorithm does not create a model of the data and by extension is a non-parametric algorithm. The value of k is more technically called a hyperparameter than a parameter, because it is a parameter of the algorithm, not a parameter associated with the data. Choosing k is important. If k is small, the algorithm tends towards low bias and high variance. As k gets larger, the algorithm tends towards high bias and low variance. Advantages:

- Makes no assumptions about the shape of the data
- Performs well in low dimensions
- Can be used for classification or regression

Disadvantages:

- Bogs down in high dimensions
- K must be chosen
- Data should be scaled for best performance
- Difficult to interpret

10.8.1 Quick Reference

Reference 10.8.1 kNN Classification

```
library(class)
predictions <- knn(train, test, cl=trainLabels, k=3)
```

Reference 10.8.2 kNN Regression

```
library(caret)
fit <- knnreg(train, labels, k=3)
predictions <- knnreg(fit, test)
```

Reference 10.8.3 Scaling and Unscaling, Method 1

```
scaled <- scale(df) # scale is built-in
# unscaled in package DMwR
unscaled_predictions <- unscale(predictions, scaled)
```

Reference 10.8.4 Scaling, Method 2

```
# train and test contain just predictors
means <- sapply(train, mean)
stdvs <- sapply(train, sd)
train <- scale(train, center=means, scale=stdvs)
test <- scale(test, center=means, scale=stdvs)
```

10.8.2 Practice to Consolidate Skills

Problem 10.1 — Practice on the Abalone Data - kNN Regression. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Abalone`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Divide the data into 80-20 train-test, setting a seed for reproducibility.
5. Normalize the data.
6. Perform knn regression with k=3. What is the cor and mse? Are these good results? Why or why not?
7. Try a range of k values to find the best k. Did the metrics improve?

Problem 10.2 — Practice on the Abalone Data - kNN Classification. Start with the same Abalone data as the previous problem, but add a size column as we did in an earlier lab.

1. Examine the rings column with range(), median(), and hist() to determine where you would like to split the data into two classes: large and small.
2. Create a new factor column for binary large/small based on the rings column and your cut-off decision.
3. Divide the data into 80-20 train-test, setting a seed for reproducibility.
4. Normalize the data.
5. Perform knn classification with all predictors except rings, using k=3. What is the accuracy of the model? Do you think these are good results? Why or why not?
6. Try a range of k values to find the best k. Did the accuracy improve?

Problem 10.3 — Practice on the Heart Data. Try the following:

1. Download the Heart data from the UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml/datasets/Heart+Disease`.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Make sure class is a factor and that all the other columns are numeric or integer.
4. Remove columns with large numbers of NAs because the knn algorithm can't handle them.
5. After removing those columns (slope, ca, and thal), reduce the heart data to only complete cases.
6. Divide the data into 80-20 train-test, setting a seed for reproducibility.
7. Normalize the predictor data.
8. Run knn() with k=3 to create predictions on the test data. What is the accuracy? Do you think this is a good result? Why or why not?
9. Try a range of k values to find the best k. What is the accuracy?

Problem 10.4 Based on your experience with the knn algorithm in R, does removing predictors necessarily improve performance? Discuss possible reasons for your answer.

Problem 10.5 If you found that for a kNN regression problem, the optimal value for k using mse differed from the optimal value using cor. Which one would you prefer? Justify your answer.

Problem 10.6 As k increases, do you think the fit to the data is more or less flexible? Justify your answer.

Problem 10.7 As k increases, do you think the fit to the data tends toward higher bias or higher variance? Justify your answer.

10.8.3 Next-Level Learning

The kNN algorithm has wide application. Data Camp has a free online course for anomaly detection using R:

`https://www.datacamp.com/courses/introduction-to-anomaly-detection-in-r`

11. Clustering

Clustering is *unsupervised* learning because we either do not have labels or choose to ignore them for the purpose of learning more about the data. The goal of clustering is to find groups within the data that share common features. Figure 11.1 shows the iris data plotted using Petal.Length and Petal.Width. The goal in clustering is to group them into homogeneous clusters.

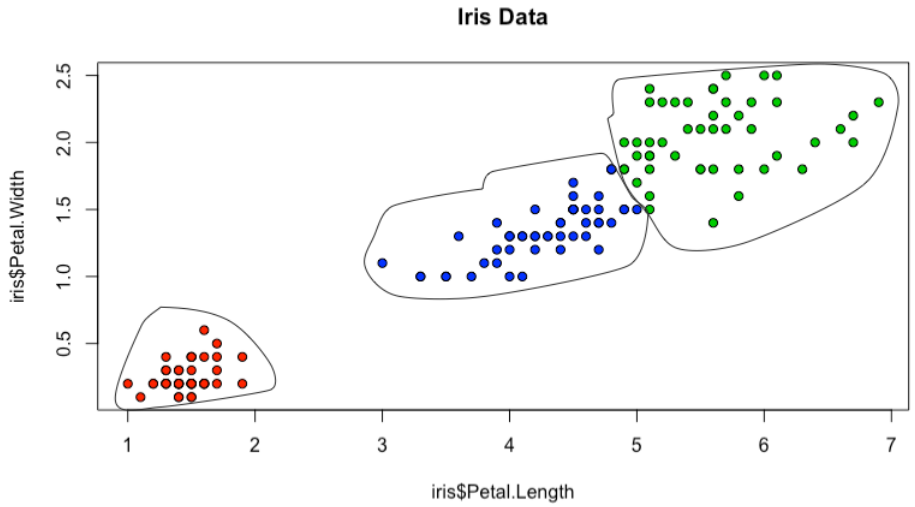


Figure 11.1: Clustering the Iris Data

11.1 Overview

There are numerous clustering algorithms, but in this chapter we focus on two of the most popular:

- k-means clustering - identifies k centers (aka centroids), and groups other observations based on nearness to the closest centroid
- hierarchical clustering - uses a distance measure to combine observations into clusters that are organized in a hierarchy

The notion of distance is central to clustering. What do we mean by distance? We can think of this as Euclidean distance but it may be optimized to be some type of variance measure. How are the clustering algorithms discussed in this chapter different from kNN? The kNN algorithm does not cluster all the observations into clusters, it simply looks for nearby observations given a test observation. The kNN algorithm is also different in that it is a supervised algorithm whereas the clustering algorithms discussed in this chapter are unsupervised.

Clustering problems have a typical workflow:

1. preprocess data
2. determine a similarity measure
3. cluster the observations
4. analyze the results
5. repeat from 2 until the clusters are meaningful

Scaling is very important in clustering because we are measuring distance (or variance). Imagine that we are clustering people by their physical characteristics, if they are 2 pounds different in weight we would consider them similar but not if they were 2 feet different in height. The number 2 has no meaning in isolation and will make sense for distance calculations only if scaled within its column. We will use R's scale() function to put all our columns in similar ranges.

11.2 Clustering in R with k-Means

The k-means algorithm is an iterative approach that starts with a random assignment. The algorithm is as follows:

1. random assignment
2. assign each observation to its closest centroid
3. recalculate the centroids
4. repeat from 2 until convergence

There are many variations on k-means. The most important variation is how step 1 is conducted. One approach is to randomly choose k observations to be the means. Another approach is to randomly assign each observation to one of k groups.

The kmeans() function is in R, we don't need to load a package. The first argument specifies the data you want to cluster. Here we are only considering Petal.Length and Petal.Width. The second argument specifies that we want 3 clusters. In the case of the iris data we know there are 3 classes so this is an unusual advantage here. Normally determining the optimal number

of clusters is done experimentally. The third argument is the number of starts. Because the algorithm starts with a random assignment, the algorithm is usually run several times with different random assignments. The kmeans() algorithm will select the best one of the 20 starts we requested. We set a seed for reproducible results but the algorithm will have 20 random starts.

Code 11.2.1 — k-Means Clustering. Iris Data.

```
set.seed(1234)
irisCluster <- kmeans(iris[, 3:4], 3, nstart=20)
irisCluster
```

The results are stored in an object named irisCluster. By typing its name at the console you get the following information.

```
K-means clustering with 3 clusters of sizes 50, 48, 52
Cluster means:
  Petal.Length Petal.Width
1     1.462000    0.246000
2     5.595833    2.037500
3     4.269231    1.342308
Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [51] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 3
      3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[101] 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 3 2 2 2
      2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2
Within cluster sum of squares by cluster:
[1]  2.02200 16.29167 13.05769
 (between_SS / total_SS =  94.3 %)
Available components:
[1] "cluster"      "centers"      "totss"        "withinss"
     "tot.withinss" "betweenss"
[7] "size"         "iter"         "ifault"
```

Notice the 3 clusters it found were 50, 48, 52 so we know that at least 2 observations ended up clustered with a different species than they truly are. That is a moot point because clustering does not have the same goal as classifying. The output also shows the cluster means. We did not scale the iris data because Petal.Length and Petal.Width, our two features, are in the same units. The within-cluster sum of squares measures the variance of the observations within each cluster. A smaller sum of squares indicates a more compact cluster. Notice that the first cluster has a very small value of 2.022. This is the compact red cluster visualized in Figure 11.1. The other clusters are more spread out as indicated by their higher sum of squares numbers of 16 and 13.

Finally, the output shows the components you can look at within the irisCluster object. For example to see the size, type `irisCluster$size` at the console. Most of this information was already supplied in the output.

11.3 Metrics

In most clustering problems we are clustering without knowing what the "true" clustering should be, or if there even is a "true" clustering. How will you know if you have a good clustering or not? Let's first look more precisely at what a cluster is. A cluster is a set of observations that satisfies the following two properties:

- Each of the n observations belongs to one of the clusters: $C_1 \cup C_2 \cup ... \cup C_k = 1, 2, ...n$
- No observation belongs in more than one cluster: $C_i \cap C_j = \emptyset$ for all i, j

One indication that we have a reasonably good clustering is if items within a cluster are homogeneous. One way to measure this is by nearness using a distance metric like Euclidean distance. That is, we average over all k clusters, the distance between every pair of points in the cluster for all features p. We want this average distance to be minimal.

$$\sum_k \frac{1}{|C_k|} \sum_{i,j \in C_K} \sum_p (x_{ip} - x_{jp})^2 \tag{11.1}$$

The term *distance* that is commonly used in connection with k-means is intuitive because we can easily visualize it from our everyday experiences. Perhaps a more mathematically convenient term would be *variance*. We want to know the variance of a point from its assigned centroid. For point x_i, we want to quantify its variance from the mean of each centroid. Each centroid has its own mean, so the squared variance from a centroid is:

$$(x_i - \bar{\mu})^2 \tag{11.2}$$

The quantity we seek to minimize is called the **within sum of squares**. This quantity can be output from the results by typing `results$withinss` at the console. This will output the within-ss for all the clusters so if you want an overall withinss, place sum() around the expression. The within-ss indicates how compact clusters are while the between-ss indicates how well separated the clusters are.

11.4 Algorithmic Foundations of k-means

Since there would be k^n ways to divide n observations into k clusters, a direct mathematical approach would be np-complete. It could not be solved in polynomial time. Further, a loss function for k-means woud be non-convex. For this reason we use an approximation algorithm. The k-means heuristic approach will give a good result but it is not guaranteed to be a global optimum. This is why we use several starts of the algorithm. By finding many local optima we hope to get close to a global optimum.

The two iterative steps of the k-means algorithm can be viewed in a more general sense as an example of the Expectation-Maximization algorithm which estimates parameters from data. The Expectation step computes the probability of the data given the parameters. The Maximization step then computes a better estimate of the parameters. The EM steps are repeated until convergence. The k-means algorithm as viewed from the EM perspective:

1. E step: each observation is assigned to the closest centroid, the most likely cluster
2. M step: the centroids are recomputed

The EM algorithm is widely used in machine learning in cases where parameters cannot be directly calculated.

Exercise 11.1 — k-means. Using the Wine Data.

Try the following:

- Load the wine_all.csv file from the github.
- Make sure type and other columns are numeric.
- Normalize the data.
- Perform k-means clustering with k=2.
- What is the correlation between the clusters and the wine type?
- Plot the data with the following code. The graph would be crowded if we used all the data so first we randomly sample 250 observations. We use pH and alcohol for the x, y axis to spread the data out. We use a golden color for the white wine and a reddish color for the red wine. The clusters are identified by triangles and circles.
- Keep in mind we are only plotting on two dimensions in the graph. The data is of what dimension? What does the clustering tell you about white and red wines in n-dimensional space?

```
j <- sample(1:nrow(wine), 250, replace=FALSE)
plot(pH[j], alcohol[j], pch=wine_clust$cluster[j],
     col=c("brown3","goldenrod1")[wine$type[j]+1],
     xlab="alcohol", ylab="pH")
```

11.5 Finding k

In unsupervised clustering, we do not know beforehand what the optimal value of k will be. We have to find it experimentally. The notebook in github provides an example of finding k using synthetic data. The rnorm() function was used to create 60 points, in 3 distributions with centers (10, 3), (27, 2) and (41, 5). These are the "true" clusters but the regions overlap a little. We plot the unclustered data with different shapes for each distribution. Then we try clustering with different values of k. Figure 11.2 shows the results of kmeans() with k=3. The shapes represent the "true" distributions. We see one purple square between the orange and purple clusters and 3 green circles between the purple and green. The bottom graph shows the results when k=4. The overlap between the purple and green in the top figure has become its own

cluster.

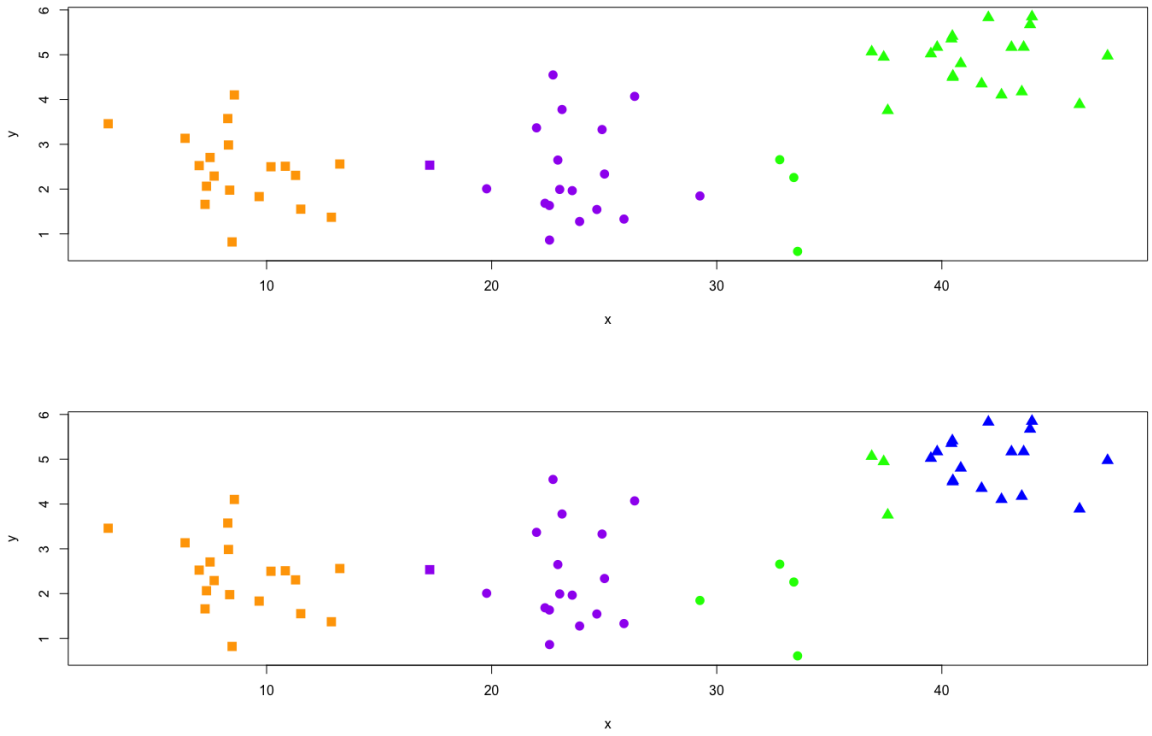


Figure 11.2: Clustering with k-means with k=3 (top) and k=4 (bottom)

We know we created the data from 3 distributions but they are not perfectly separated clusters. In the online notebook we tried k=2, 3, 4, and 5, and printed the withinss:

```
[1] "k=2:  2530.74905628694"
[1] "k=3:  611.695448018061"
[1] "k=4:  373.0421592289"
[1] "k=5:  284.426492959912"
```

It seems there is a dramatic drop from k=2 to k=3 then it gradually decreases. It makes sense that the larger the number of clusters, the smaller the within-ss. After all, if k=n (each observation in its own cluster) then within-ss would be 0.

Rerunning the algorithm with various values of k is tedious, let's do that in a function to try k = 2 to 9. The function below reruns kmeans() with each value of k in 2:9, keeping track of the sum of the within-ss from the results. The plot is shown in Figure 11.3. Notice that within-ss is decreasing but that there is an "elbow" at k=3. The value of k at the elbow is chosen for k because k values higher than that are just reflecting the observation that larger numbers of clusters result in smaller clusters and consequently lower within-ss.

Code 11.5.1 — Finding k. Using a Function.

```
plot_withinss <- function(df, max_clusters){
  withinss <- rep(0, max_clusters-1)
  for (i in 2:max_clusters){
    set.seed(1234)
    withinss[i] <- sum(kmeans(df, i)$withinss)
  }
  plot(2:max_clusters, withinss[2:max_clusters], type="o",
      xlab="K", ylab="Within Sum Squares")
}
plot_withinss(df, 9)
```

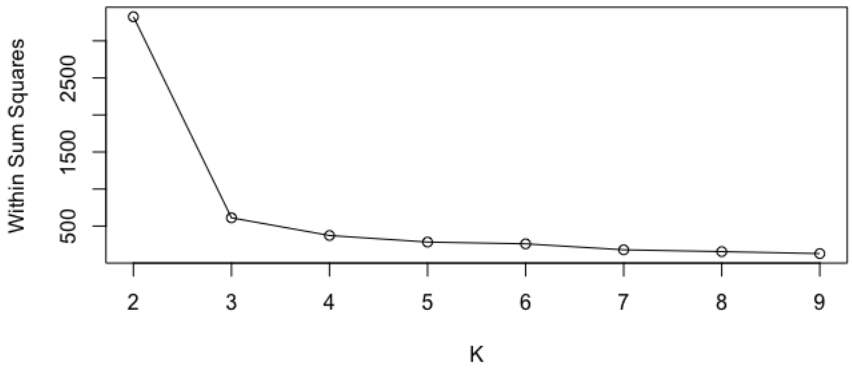


Figure 11.3: Finding K

11.5.1 NbClust

We wrote a simple function above to help us determine the best value for k. There is a very sophisticated function in package `NbClust` that provides information on over 30 indices. The online notebook uses NbClust to confirm that the best k is 3. Figure 11.4 shows that k=3 was selected by the majority of the criteria.

There are dozens of clustering metrics and that topic is beyond the scope of this book. We will briefly discuss one clustering metric that is available via the NbClust package. The *Rand index* is a measure of cluster purity that ranges from -1 to +1. Rand Index close to 0 means that the clustering was random. The Rand Index for k=3 was very high, 0.92 which is another indication that k= 3 is the best choice.

The graph in Figure 11.4 is a bar graph that indicates how many metrics vote k=n as the best clustering, out of dozens of metrics. The fact that the bar at k=3 is several times taller

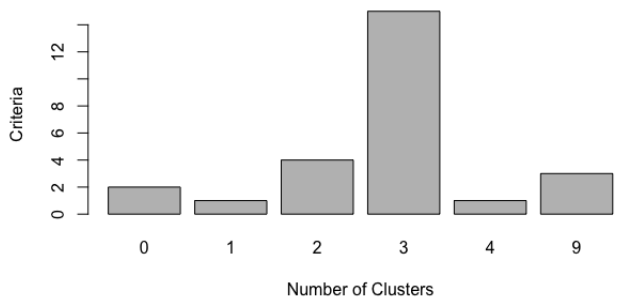


Figure 11.4: Finding K with 30 Indices in NbClust

than the other bars does not mean that a clustering with k=3 will be several times better than other values for k. It simply means that more metrics indicating that k=3 was the best choice.

11.6 Hierarchical Clustering

As we have seen, one of the difficulties of using k-means is that we have to specify k beforehand and it takes quite a bit of exploration to find the optimal k. Hierarchical clustering is a very different type of clustering. We do not have to specify how many clusters we have before running it. Another unique thing about hierarchical clustering is that it creates a dendogram of the clustering, which you can see in Figure 11.5.

A notebook in the github provides an example of hierarchical clustering with the nutrient data set in package `flexclust`. This data set lists the energy, protein, fat, calcium and iron for 27 prepackaged foods. Since the 5 variables are in different units, they are scaled first. R has a built-in dist() function that calculates Euclidean distance by default for a matrix. Once we have the distance, we input this into the hclust() function, built into R. Then the dendogram is plotted.

Code 11.6.1 — Hierarchical Clustering. Nutrient Data.

```
d <- dist(nutrient.scaled)
fit.average <- hclust(d, method="average")
plot(fit.average, hang=-1, cex=.8,
     main="Hierarchical Clustering")
```

11.6.1 The Algorithm

There are many variations of hierarchical clustering, here is the "bottom-up" version:

1. Place each observation in its own cluster
2. Calculate the distance between each cluster and every other cluster
3. Combine the two closest clusters

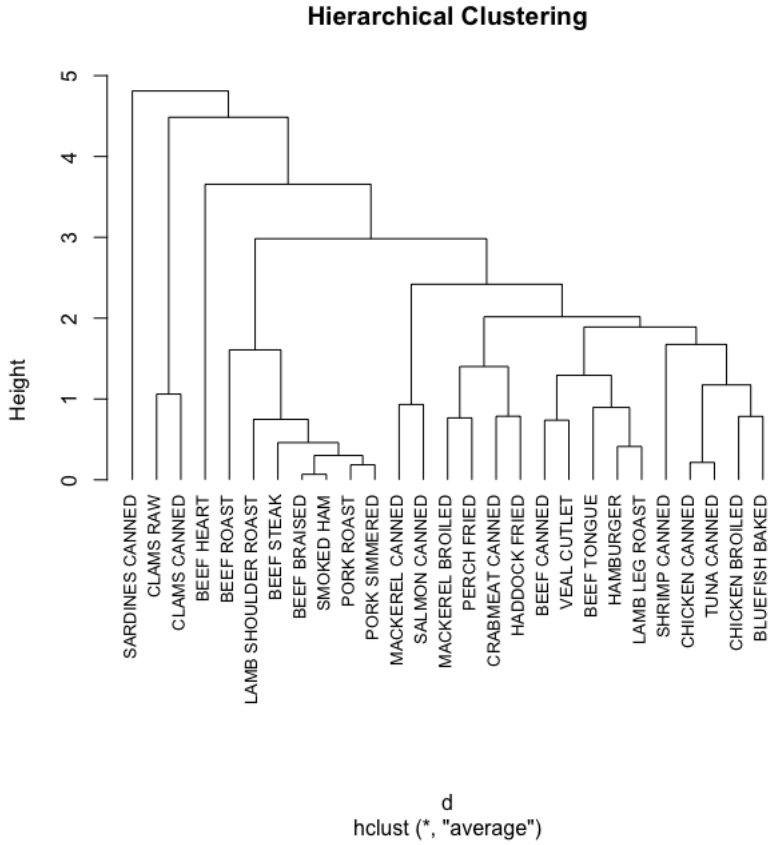


Figure 11.5: Hierarchical Clustering on Nutrient Data

4. Repeat steps 2 and 3 until all clusters merge into one cluster

How do you measure distance between clusters? There are 3 different types of measurements, called linkage, that we can use in hierarchical clustering:

1. single linkage: the shortest distance between any points in the clusters; tends to create elongated clusters
2. complete linkage: the longest distance between any points in the clusters; more compact but sensitive to outliers
3. average linkage: the average distance between points in the clusters

11.6.2 Cutting the Dendogram

Figure 11.6 shows the dendogram with 3 possible cuts in colored lines. The blue (topmost) line would result in two clusters, while the red and green lines would result in 3 and 5 clusters, respectively. The built-in cutree() function can be used to cut a tree created by hclust(). The red (middle) line cut in Figure 11.6 could be created by command cutree(fit.average, 3), where fit.average is the hierarchical clustering and 3 is the number of clusters to retrieve. The cutree() function returns a vector of indices of the cluster to which each observation belongs.

The online notebook for hierarchical clustering shows how to loop through possible cuts

and evaluate the clusters. After visually inspecting the clusters to see if they make sense for the task at hand, you might consider printing tables of the clusters as shown in the online notebook. Another option is to use some kind of clustering metric.

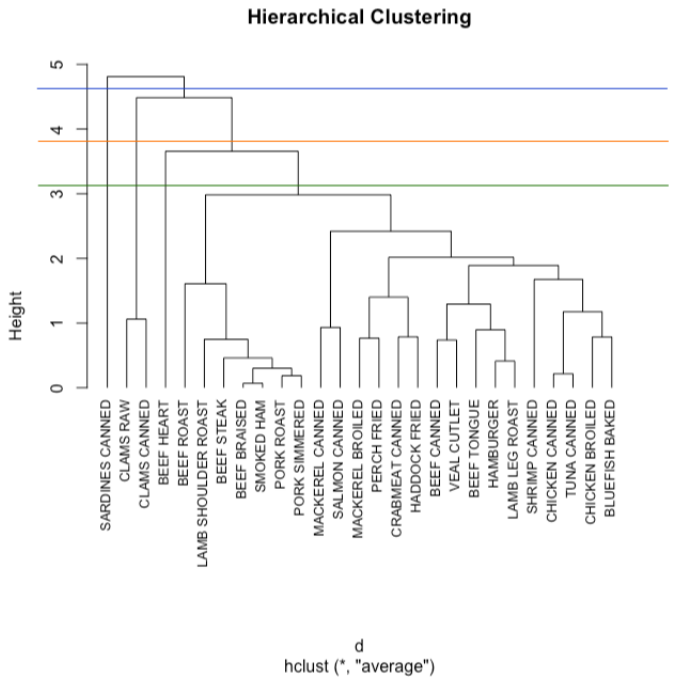


Figure 11.6: Cutting a Dendogram

11.6.3 Advantages and Disadvantages of Hierarchical Clustering

Hierarchical clustering tends to bog down with a lot of data. Another potential disadvantage is that it is a greedy algorithm. Once an observation is in a cluster, it is stuck there. The algorithm does not try several starts as k-means does.

So when should you use hierarchical clustering, also called hierarchical agglomerative clustering? If you suspect there is some hierarchical structure in data, then this algorithm may find it.

Exercise 11.2 — Hierarchical Clustering. Using the same normalized wine data as above, try the following:

- Perform hierarchical clustering on the entire wine data. Describe the dendogram.
- Randomly sample 10 observations and perform hierarchical clustering. Compare observations that were clustered together to identify any patterns.

▪

11.7 Summary

Although data exploration is an important first step in any machine learning approach, it is vitally important in clustering. You have to get to know your data before you can begin to select features for clustering and in the end to evaluate your results. Otherwise you are just seeing elephants in clouds.

The algorithms explored in this chapter are examples of unsupervised machine learning, specifically clustering algorithms. How are these techniques used? Clustering is helpful when a lot of data is available but it is not labeled. Clustering can find patterns in the data that would be difficult for humans to find. Use case examples include: finding customer clusters for marketing, finding clusters of home types and values for city planning, clustering species in biology, identifying high crime areas in police work, and much more.

11.7.1 Quick Reference

Reference 11.7.1 kmeans

```
# scale df first if necessary
# specify k=3 and  nstart=25
results <- kmeans(df, 3, nstart=25)
# examine results
results
```

Reference 11.7.2 Hierarchical

```
# scale df first if necessary
d <- dist(df)
fit.average <- hclust(d, method="average")
# plot the dendogram
plot(fit.average, hang=-1, cex=.8,
     main="Hierarchical Clustering")
```

11.7.2 Practice to Consolidate Skills

Problem 11.1 — Practice k-means on the Wine Data. Try the following:

1. Use the wine data referenced in this chapter, which is available on the github.
2. We saw in the chapter how the wine type, which is a factor being treated as an integer, pulled the data into two clusters. This is not helpful in learning more about the data. For this exercise, remove the red/white type column.
3. Normalize the data.
4. Create a plot of within-ss to help determine the best k.
5. Perform k-means with this value.

Problem 11.2 — Practice Clustering on the Glass Data. Try the following:

- Load the Glass data set from package mlbench. Research this data set and write a brief description of the columns.
- Run kNN with k=7 on the unscaled data to predict glass type from the other columns.

- Now scale the data and see if your performance improves or not.
- Try different values of k to see if you find higher accuracy.
- Run the k-means algorithm with k=7. Comment on the result of the clustering. Make a table of values like this: `table(df$Type, glass_cluster$cluster)` and comment on any patterns you see.
- Visually check how homogenous the clusters are by printing the clusters and types of glass in each cluster.
- Use the plot_withinss function from this chapter to find the best cluster on the data.
- Use the NbClust() function to find the suggested value of k. Cluster with this value.
- Compare the Rand index values for different clusterings that you have tried.
- Perform hierarchical clustering on the data. Try cuts at 3, 4, 5, and 6, comparing the Rand index at each.
- Comment on which cut you think is best, and why.

Problem 11.3 How is the kNN algorithm different from k-means?

Problem 11.4 Contrast the meaning of "k" in kNN, k-means, and k-fold cross validation.

Problem 11.5 Is the k-means algorithm guaranteed to find the optimal clustering? Why or why not?

Problem 11.6 Earlier it was stated that hierarchical clustering is a greedy algorithm. What does this mean, and why is it a potential disadvantage?

11.7.3 Next-Level Learning

The book *Practical Guide to Cluster Analysis in R* by A. Kassambara is partially available here: http://www.sthda.com/

The site also has a page devoted to hierarchical clustering: `http://www.sthda.com/english/wiki/print.php?id=237`

12. Decision Trees, Random Forests

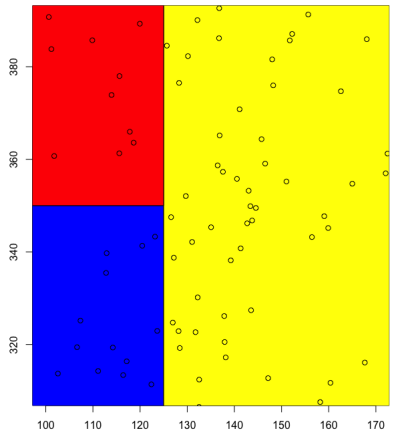


Figure 12.1: Decision Regions

12.1 Overview

Decision trees recursively split the input observations into partitions until the observations in a given partition are uniform. These regions will be rectangular in two dimensions as seen in Figure 12.1 and cuboid in higher dimensions. Notice that the decision boundaries are linear,

and aligned with the axes. The algorithm is greedy and does not go back and reconsider earlier splits. Once the vertical split at about 125 is made in the figure below, the only considerations at that point is whether or not to further divide the regions to the right and to the left. Decision trees have been around since the 80s and can be used for either classification or regression. They are not among the best performing algorithms but have the advantage that they are highly interpretable and give insight into the data.

12.2 Decision Trees in R

There are several packages with decision tree functions in R, among them rpart and tree. The latter seems to be a little more robust and prints better trees so we will use that for our examples. However, the online example uses both rpart and tree so you can compare. Figure 12.2 shows the output decision tree, which has 6 leaf nodes.

Code 12.2.1 — Decision Tree. Iris Data

```
library(tree)
tree_iris2 <- tree(Species~., data=iris)
plot(tree_iris2)
text(tree_iris2, cex=0.5, pretty=0)
```

If you type the tree name at the console you get a tree representing branches by indendation. This also includes the split data at each level. The summary() function on the model also gives some additional information. In the online notebook we also ran the algorithm with the same train set we have used before with iris and got 3 misclassified observations for a mean accuracy of 94%, but then we know that is typical for the Iris data set, having run it through many algorithms.

Exercise 12.1 — Decision Tree. Using the Wine Data.

Try the following:

- Load the wine_all.csv data and divide it into 80-20 train and test.
- Perform logistic regression to predict type from all other predictors.
- Perform knn on a scaled version of the data, again predicting type.
- Create a decision tree and evaluate its results. Note the Quick Reference at the end of the chapter which shows how to use predict() for decision trees.
- Print your tree.
- Compare the results of the 3 algorithms and consider the advantage and disadvantages of each.

■

12.3 The Algorithm

Since decision trees can be used for regression or classification we will look at these separately, beginning with regression. Both create recursive binary splits.

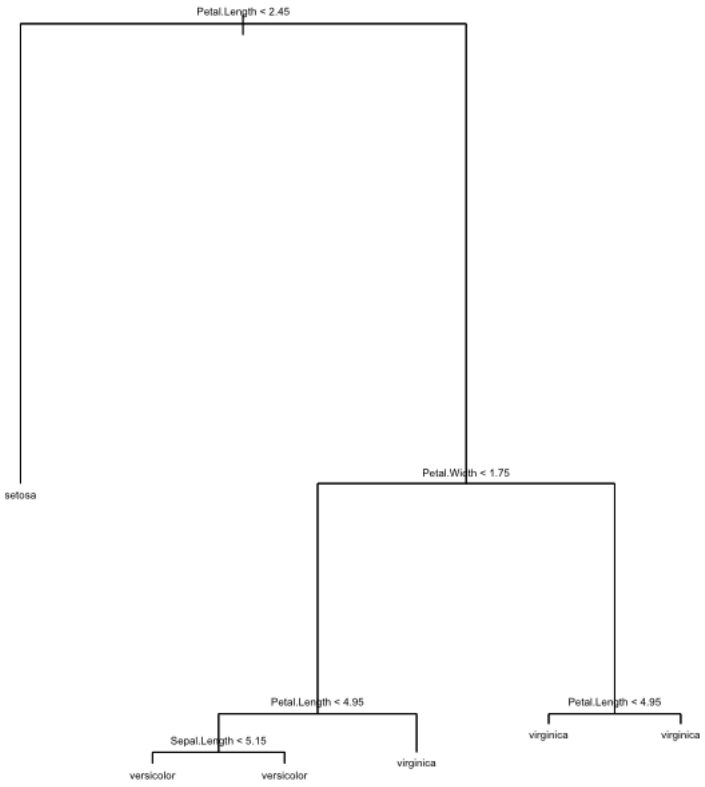


Figure 12.2: Decision Tree for Iris

12.3.1 Regression

In linear regression our goal was to minimize RSS over all the data. In decision trees, we want to minimize RSS within each region, where $\hat{y}_r$ is the mean response for the training observation in region r.

$$\sum_{r=1}^{R} \sum_{i \in r} (y_i - \hat{y}_r)^2 \tag{12.1}$$

This is computationally infeasible in part because the data space could be divided into virtually infinite regions. Instead, a top-down, greedy approach is used to partition the data. To start, all predictors are examined to see if they make good splits in the data, and for each predictor the numerical value at which to split must be determined. Our first split will divide

the data into regions r1 and r2. We will consider predictor X_1 which has a split value $split_1$, such that observations in r1 are less than the split value and other observations are in r2.

$$RSS_{X_1} = \sum_{i \in r_1} (y_i - \hat{y}_{r1}) + \sum_{i \in r_2} (y_i - \hat{y}_{r2}) \tag{12.2}$$

This splitting process is repeated until a stopping threshold is reached.

12.3.2 Classification

The same recurvise, top-down, greedy algorithm is used for classification except that RSS is replaced by counts of classes in regions. A measure such as accuracy is not sufficient for splitting regions so other measures such as entropy or the Gini index are used. These are discussed in the mathematical foundations section. The goal of either metric is to measure the purity of the regions, how homogenous they are.

12.3.3 Qualitative data

The discussion of the algorithm for regression and classification assumed quantitative data. When splitting qualitative data, a binary feature would make a natural split between the two values. Features with more than 2 values would assign one or more values to one branch and the remaining ones to the other. Decision trees can handle qualitative data without having to create dummy variables.

12.4 Mathematical Foundations

In this section we give further information about entropy, information gain, and the Gini index. We start with entropy, which measures uncertainty in the data, and information gain, which measures reduction in entropy.

12.4.1 Entropy

First we are going to talk about entropy, a measure of uncertainty in the data. The formula for entropy is given below. H() is the traditonal symbol for entropy. Notice that the probability is multiplied by the log of the probability for each class. These are summed and then negated.

$$H(data) = -\sum_{c \in C} p_c log_2 p_c \tag{12.3}$$

The probability of a class is just the fraction of data observations in that class. Let's look at the tennis data set as a simple example for these calculations.

The entropy of the entire data set is determined by counting 9/14 days when we did play and 5/14 when we did not:

$$H(tennis) = -\frac{9}{14} \times log_2 \frac{9}{14} - \frac{5}{14} \times log_2 \frac{5}{14} = .94 \tag{12.4}$$

day	outlook	temp	humidity	wind	play
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 12.3: Tennis Data Set

How do we interpret entropy? The x-axis in Figure 12.4 indicates the probability. At .5 probability, entropy is maximized. There is a lot of confusion in the data, it is the opposite of homogenous. At higher or lower probabilities, we see that entropy is lower. Lower entropy means the data is more homogenous. The probability of play in the tennis data is 0.64 and our entropy was .94, very high.

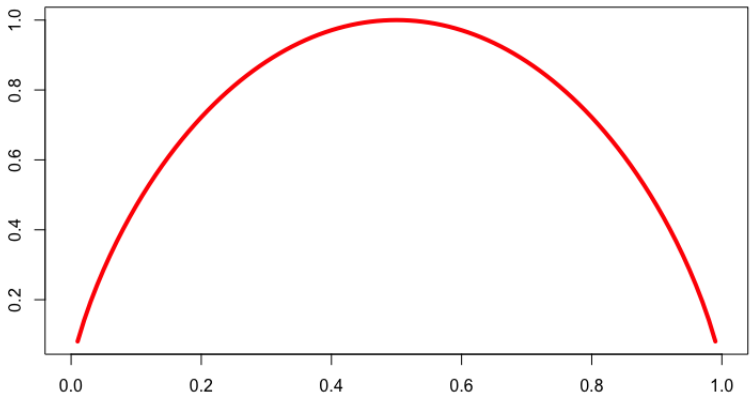


Figure 12.4: Entropy

Now let's say we want to split on wind. Wind has two levels: weak and strong. When wind is weak we have 6 examples where we play and 2 where we do not. Let's compute the entropy.

$$H(wind = weak) = -\frac{6}{8} \times log_2 \frac{6}{8} - \frac{2}{8} \times log_2 \frac{2}{8} = .811 \tag{12.5}$$

What is the entropy of wind=strong? Since we have an even split play=3 and not-play=3 we know our entropy is 1. In order to compute the entropy of wind we need to combine the

entropy of wind=weak and of wind=strong in a formula that weights it by the relative size of these two groups, 6/8 and 2/8.

12.4.2 Information Gain

Information gain is the reduction in uncertainty we would have, given a certain choice. It is the starting entropy minus the entropy of the split. In our case, it is the entropy of the entire data set (.94) minus the weighted entropy of wind. The formula is shown below, where f represents each value of a feature and N is the total count of items.

$$IG(split) = H(data) - \sum_{f} \frac{|f|}{N} H(split_f) \tag{12.6}$$

$$IG(split_{wind}) = .94 - 8/14 * .811 - 6/14 * 1 = 0.048 \tag{12.7}$$

Running similar IG statistics reveals that IG(outlook) = 0.246, IG(humidity) = 0.151, and IG(temperature) = 0.029. It turns out that outlook has the highest IG, so we would choose outlook for the decision tree split.

12.4.3 Gini index

For a given region, for each class in the region, Gini measures homogeneity. There are many variations of this formula, here is one:

$$Gini = 1 - \sum_{k \in K} p_k^2 \tag{12.8}$$

$Gini(tennis) = 1 - (9/14)^2 - (5/14)^2 = 0.459$
The maximum Gini index is 1.0. Let's look at the Gini index for wind.
$Gini(wind = strong) = 1 - (3/6)^2 - (3/6)^2 = 0.5$
$Gini(wind = weak) = 1 - (6/8)^2(2/8)^2 = .375$
$Gini(wind) = 6/14 * 0.5 + 8/14 * .375 = .43$

We weight by the proportion of observations in each class to get a Gini for wind of 0.43, not that great. The Gini index will give comparable results to information gain, but we want a minimal Gini and a maximum IG.

12.5 Tree Pruning

Decision trees are highly sensitive to variations in the data. For this reason if we grow decision trees out fully on a training set they are likely to overfit. One strategy to reduce overfitting is to grow the tree all the way out, then prune it back to get a subtree. T_0 represents the fully grown tree. There are $|T|$ subtrees in which some internal node has been collapsed to combine

its children into one region. Let Q represent the criterion for tree building for either regression or classification. Then subtrees can be indexed by lambda:

$$Subtree_t = Q + \lambda|T| \tag{12.9}$$

The optimal subtree can be found by cross validation or using a held out validation set. If lambda = 0 our subtree is the original tree. As lambda increases, the subtree is smaller and less tuned to the training data.

There is no guarantee that a pruned tree will generalize better to new data for improved performance. However the pruned tree may be more interpretable since it should have fewer branches.

The code below shows how to use cv.tree() to try various trees. We plot the deviance by tree size. It looks like the best tree is 5. Deviance is lower at a size of 8 but we are concerned that this may overfit the data. After performing cross validation, we can prune it with the prune.tree() function, and print the pruned tree.

Code 12.5.1 — Decision Tree. Cross validation.

```
cv_tree <- cv.tree(tree1)
plot(cv_tree$size, cv_tree$dev, type='b')
tree_pruned <- prune.tree(tree1, best=5)
plot(tree_pruned)
text(tree_pruned, pretty=0)
```

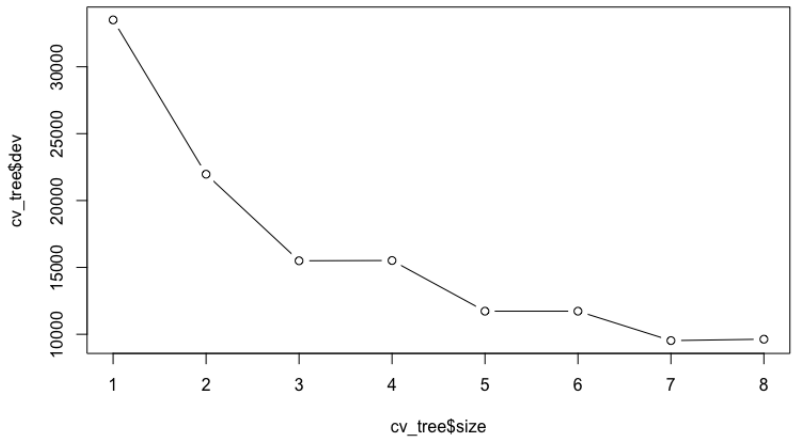


Figure 12.5: Cross Validation on Decision Tree

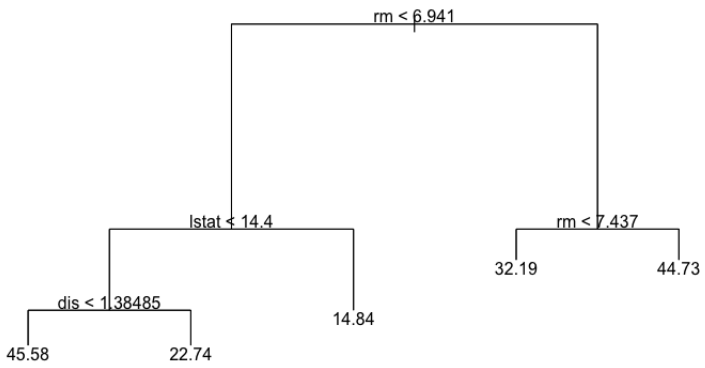


Figure 12.6: Pruned Decision Tree

Exercise 12.2 — Pruning a Decision Tree. Using the Wine Data.

Try the following:

- Using the cross-validation technique demonstrated in this chapter, prune the decision tree you created for the wine data.
- Print the tree.
- Evaluate your results on the same test data as before.
- How does accuracy compare with the unpruned tree? Which tree do you prefer, and why?

12.6 Random Forests

The term bootstrap means to repeatedly sample from a data set. Bagging is short for bootstrap aggregation and the main idea is to repeatedly sample the data to overcome the variance in a technique. As we have discussed, decision trees have high variance. The random forest builds on the idea of bagging but the trees are de-correlated as follows. At each split, a random subset of predictors is selected from all the predictors and one is chosen. A typical size of the subset is the square root of the number of predictors. Suppose there is a strong predictor that many bagged trees would choose first. The random forest approach prevents trees from chosing the same predictors in the same order. This allows new trees to be discovered that may outperform trees choosing the strongest predictor first. In effect, the down side of the greedy algorithm is mitigated by restricting the choice of predictors.

12.7 Cross validation, Bagging, Random Forests in R

In the github is an example of a regression tree on the Boston housing market. For comparison purposes we first built a linear regression model using all predictors. The correlation of the predicted versus the test median home value was .8 and the root mean squared error was 5.36, meaning that the home price was off by about $\$5,360$. Running the tree algorithm on the same train/test data had a correlation of .88 and an rmse of 4.2. The decision tree got better results than linear regression. The following code example shows how to perform cross validation on the tree. The output graph in Figure 12.5 indicates that a tree size of 5 should give good results, perhaps without overfitting. Next the code shows that we prune the tree. The pruned tree is shown in Figure 12.6. The pruned tree has 5 leaf nodes compared to 8 in the unpruned tree which you can view online. The pruned tree had a correlation of 0.81 and an rmse of 5.27. This shows a reduced performance but has the advantage of being more interpretable with fewer leaf nodes.

Next we try bagging and random Forest on the same data. The parameter `importance` tells the random Forest algorithm to consider predictor importance. The parameter mtry specifies the number of variables to sample at each split. The default value for classification is sqrt(p) and for regression is p/3, where p is the number of predictors. If mtry is set to the number of predictors, then bagging is performed instead of the Random Forest. In the code block below, a Random Forest is created because mtry is left to the default values.

We see in the output, shown below the code example, that 500 trees were created, trying 4 variables at each split. The test correlation on the bagged trees jumped to 0.93 and the rmse decreased to 3.4, meaning that we were off on the home price by only an average of $\$3,400$.

Code 12.7.1 — Random Forest. Let mtry use default settings.

```
library(randomForest)
set.seed(1234)
rf <- randomForest(medv~., data=train, importance=TRUE)
```

```
Call:
 randomForest(formula = medv ~ ., data = train, importance = TRUE)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 4

          Mean of squared residuals: 10.66999
                    % Var explained: 87.5
```

In the online notebook you can see that we also tried bagging by setting the number of variables to the maximum. The results were very similar. Recall that the difference between bagging and Random Forest is that the Random Forest is discouraged from predicting the strongest predictor first so that the trees are more unique. In the Boston data, rm was the strongest predictor on all the models so it seems that choosing this as the first split is the best choice. The strongest predictor may not always be the best initial split, however.

Exercise 12.3 — Bagging and Random Forest. Using the Wine Data.

Try the following:

- Using the unpruned decision tree you created earlier, try bagging.
- How many trees were created?
- The results from bagging talk about an OOB estimate of error. Research what that means and write a 1-2 sentence explanation.
- Now try random forest.
- Compare the output of the bagging and the random forest. What do you observe?
- Compare the results of bagging and random Forest to your earlier results in terms of accuracy.
- Besides accuracy, what other considerations would cause you to prefer one algorithm over the others?

12.8 Summary

As you can see in the Boston example online, linear regression and the regression tree got fairly similar results. Linear regression will typically outperform decision trees when the underlying function is linear. However if the relationship between the predictors and the target is not linear and complex, decision trees will probably perform better. In the Check Your Understanding examples, you observed that decision trees can perform similarly to logistic regression, particularly with bagging or random forests.

Decision trees are considered to be non-parametric algorithms. Decision trees have an advantage in being easily interpretable. Decison trees have low bias and high variance. Decision trees are sensitive to the distribution of predictors so slightly different data can result in very different trees. This high variance can be overcome with bagging or random forests. Pruning the tree may help it generalize to new data by holding back from overfitting but this is no guarantee of improving performance.

12.8.1 Quick Reference

Reference 12.8.1 Build Decision Tree

```
library(tree)
tree1 <- tree(y~., data=df)
plot(tree1)
text(tree1, cex=0.5, pretty=0)
```

Reference 12.8.2 Evaluate Decision Tree

```
# remove type="class" for regression
tree_pred <- predict(tree1, newdata=test, type="class")
table(tree_pred, test$type)
mean(tree_pred == test$type)
```

Reference 12.8.3 Cross Validation and Pruning

```
# cross validate to find  best
cv_tree <- cv.tree(tree1)
plot(cv_tree$size, cv_tree$dev, type='b')

# prune to best
tree_pruned <- prune.tree(tree1, best=5)
```

Reference 12.8.4 Bagging or Random Forest

```
library(randomForest)
set.seed(..)
# importance=TRUE considers variable importance
# mtry=p (number of predictors)
#      will perform bagging instead of random forest
tree_bag <- randomForest(y~., data=df, importance=TRUE, mtry=13)
random_forest <- randomForest(y~., data=df, importance=TRUE)
```

12.8.2 Practice to Consolidate Skills

Problem 12.1 — Practice Decision Trees on the Abalone Data. Try the following:

1. Use the abalone data available on the github. Load the data, providing names as done in the labs for Chapter 4.
2. Divide the data into train and test sets.
3. Perform linear regression and evaluate correlation and mse on the test set.
4. Perform knn regression and compute correlation and mse. You will need to convert sex from a factor to integer before scaling. Copy your train and test sets to a new name so the original train and test are preserved.
5. Create a decision tree, print the tree, and evaluate on the test data.
6. Try bagging and compute your metrics on the test data.
7. Compare the results on all models and discuss why you think the best model was able to outperform the others.

Problem 12.2 — Classification on the Heart Data. Try the following:

- Load the processed.hungardian.data.csv from the git hub. Make sure class is a factor.
- Remove columns 11:13 because they have too many NAs, and use complete.cases() to get rid of remaining NAs.
- Split the data into 80-20 train and test.
- Create a logistic regression model on the data and evaluate its accuracy on the test data.
- Create an additional train and test set that are normalized for knn.
- Run knn with k=10 and evaluate its performance on the test data.
- Create a decision tree, print it, and evaluate its performance.
- Try the random forest algorithm and evaluate its performance.
- Comment on which algorithm performed best.

Problem 12.3 — Classification on the Glass Data. Try the following:

- Load the Glass data set from package mlbench. Research this data set and write a brief description of the columns.
- Divide the data into 80-20 train-test.
- Create scaled versions of the train and test data for kNN. Try various values for k and see how accurate the algorithm can get.
- Create a decision tree, print it, and evaluate its accuracy on the test data.
- Try random forest on the data and evaluate its accuracy.
- Comment on which cut you think is best, and why.
- Discuss what we would have to do with the data to use logistic regression.

Problem 12.4 Compare how linear regression makes predictions compared to decision trees.

Problem 12.5 Compare how kNN makes predictions compared to decision trees.

Problem 12.6 Explain why linear regression has high bias and decision trees have high variance.

12.8.3 Next-Level Learning

Professor Lior Rokach from Tel-Aviv University has shared his overview of decision trees here: `http://www.ise.bgu.ac.il/faculty/liorr/hbchap9.pdf`

13. The Craft 4: Feature Engineering

13.1 Feature Engineering

Feature engineering involves changing the data in some way to improve performance on the algorithm. Let's say we have a date field and we know that our target variable has different values for weekdays versus weekends. Some algorithms may be able to work with such data but other algorithms may perform better if we change the date field as a binary weekday/weekend factor. If we have columns in the data that are highly correlated with each other, we might consider deleting all but one of the correlated variables to reduce the total number of columns fed into the algorithm.

13.1.1 Principal Components Analysis

Principal Components Analysis (PCA) is a data reduction technique that can help you reduce the dimensions of your data. PCA transforms the data into a new coordinate space while reducing the number of axes. Each axis of the reduced space represents a principal component. The first PC represents the dimension of greatest variance, and the other PCs represent decreasing variance. An intuitive explanation is to visualize n-dimensional space reduced to a k-dimensional elliptical space where k is smaller than n. This is a data reduction technique, we are losing data and may correspondingly lose accuracy in any models.

We are going to take a quick look at PCA on the iris data. In the notebook in github, we first divide the iris data into train and test, 100 and 50 observations respectively. Then the preProcess() function from caret is used to find the principal components.

Code 13.1.1 — PCA. Iris data.

```
pca_out <- preProcess(train[,1:4],method=c("center","scale","pca"))
pca_out
```

```
> pca_out
Created from 100 samples and 4 variables

Pre-processing:
  - centered (4)
  - ignored (0)
  - principal component signal extraction (4)
  - scaled (4)

PCA needed 2 components to capture 95 percent of the variance
```

We see that PCA reduced the 4 variables to 2 principal components. These two components capture 95% of the variance in the data. Let's plot the two components PC1 and PC2 and color the points according to the true class. We see that we have one class that is separate from the other two, shown in red. And we see that the blue and green are fairly well seperated but that there is some overlap. This suggests that classification will be possible but challenging.

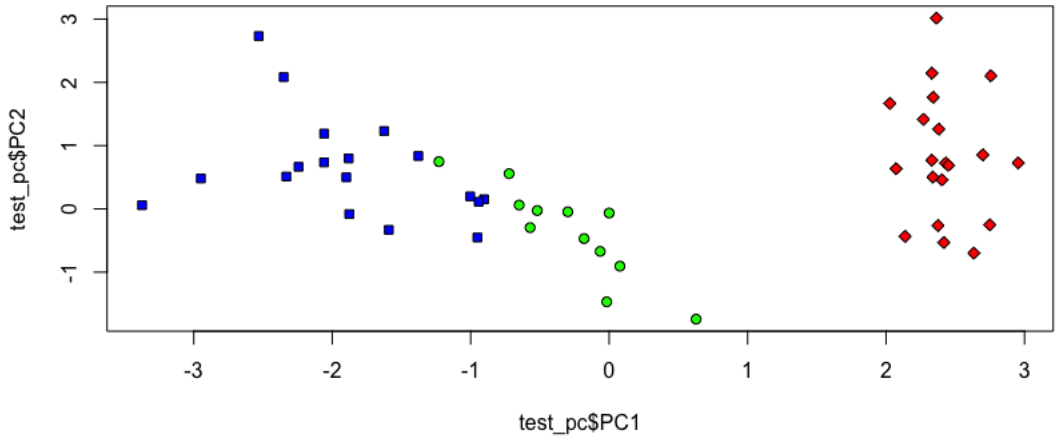
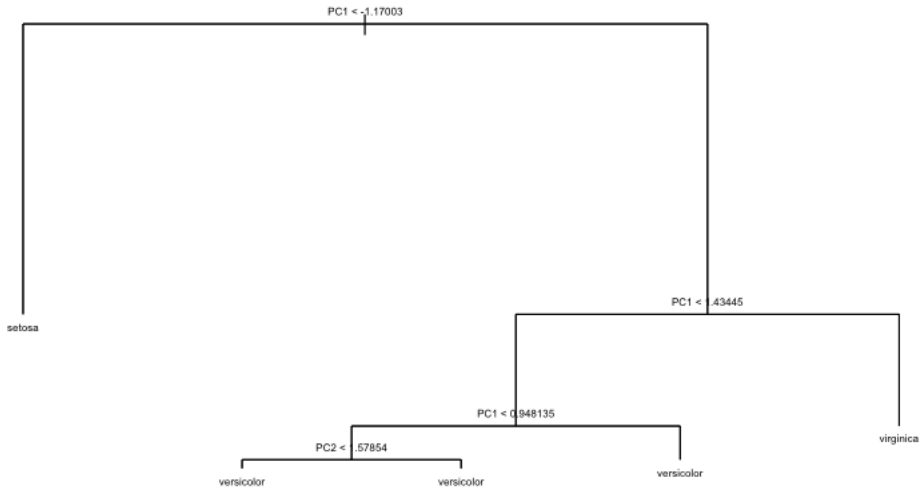


Figure 13.1: Principal Components for the Iris Data

In the online notebook, we tried kNN and got a mean accuracy of .94. Although this is lower than the 98% accuracy we got in Chapter 9 using knn on all 4 predictors, it is a confirmation that PCA is capturing something important in the data. We also ran a decision tree on the PCA data and got a little lower accuracy of .92. The tree is shown below.



lda

Figure 13.2: Principal Components Decision Tree

13.1.2 Linear Discriminant Analysis

PCA is applied to data without regard to class and such is used like other unsupervised methods for data analysis. Linear Discriminant Analysis, LDA, does consider the class. LDA seeks to find a linear combination of the predictors that will maximize the separation of the classes while minimizing the within-class standard deviation. There is an lda() function in package MASS which we use on the same data as PCA above.

Code 13.1.2 — Linear Discriminant Analysis. Iris Data.

```
library(MASS)
lda1 <- lda(Species~., data=train)
lda1$means
```

```
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa         4.943333    3.350000     1.470000   0.2633333
versicolor     5.973684    2.752632     4.268421   1.3315789
virginica      6.515625    2.925000     5.493750   1.9750000
```

We see that the LDA analysis has identified means for all variables by class. When we used the lda1 model for prediction in the online notebook, we got an accuracy of 98%. Further when we plot the lda predictions by class using the two linear discriminants returned by the model, we see a nice separation. The color of the points indicates the true class while the shape indicates the predicted class.

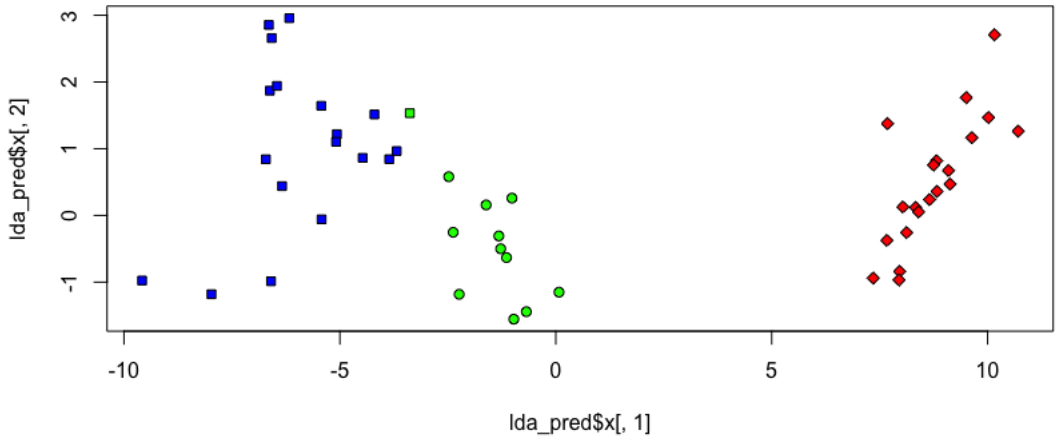


Figure 13.3: Linear Discriminant Analysis, Iris Data

In comparing PCA and LDA, the most significant difference is that PCA is unsupervised and LDA is supervised. Both can be used for dimensionality reduction. In this example, both reduced the 4 iris predictors to two predictors. LDA will have an advantage over PCA when the class is known.

IV Part IV: Modeling the World

Preface to Part Four

In Part 8 we look at algorithms from the field of AI that are used to model probabilistic variables, and/or environments. These methods do not fit into either the unsupervised or supervised branches of machine learning, and more properly fit into an AI book than a machine learning one.

- Bayesian Networks are graphical representations of conditional dependencies in data. Since the late 1980s they have been used in targeted domains such as medicine and economics where expert knowledge and Bayesian techniques combine for synergistic effect.
- Markov Models consists of states and transition probabilities that are linked together in a chain. In a Markov process, history has no meaning. All that matters is the current state and the transition probabilities to other states.
- A Markov Decision Process is built upon a Markov chain, but additionally has an agent that learns over time, thus changing the transition probabilities as it learns.
- Reinforcement Learning builds upon a Markov Decision Process by letting an agent learn not only the transition probabilities but learn about the states as well.

14. Bayes Nets

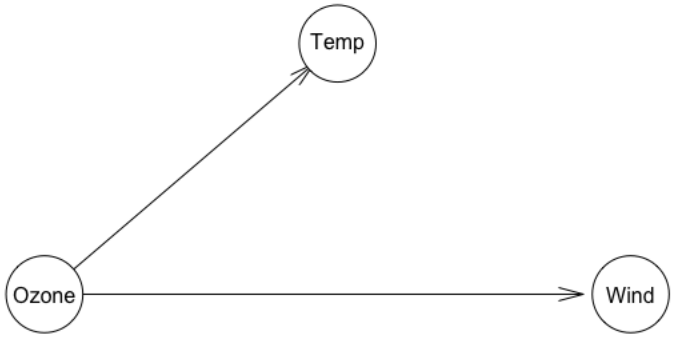


Figure 14.1: Airquality Bayesian Network

Bayesian networks, sometimes called belief networks, are unique in that they can be customized to encode not only Bayesian probabilities but also expert human knowledge. In turn, the network itself is highly interpretable. Figure 14.1 shows that Ozone is conditioned

on Temp and Wind. Ozone is the parent node, Temp and Wind are child nodes. The absence of links can convey information as well. Notice there is no link between Temp and Wind. In a Bayes' network there are no target or predictor variables. Each node represents a random variable and arrows represent their probabilistic dependencies. A very important thing to keep in mind with Bayesian networks, is that the probabilistic dependencies should not be interpreted as causation. Rather, the algorithm is identifying correlations, not necessarily causations, in the data.

14.1 Bayes Nets in R

A Bayesian network is a directed acyclic graph as seen in Figure 14.1. The network was created with the code below. The bn.fit() function used below needs all the variables to be factors so first we subset the airquality data, removed rows with NAs, and converted them to binary factors with 1 meaning high and 0 meaning low. The text below the code example shows the conditional probabilities of the fit model.

Code 14.1.1 — Bayesian Network. Airquality data.

```
library(bnlearn)

f <- function(v){
  m <- mean(v)
  factor(ifelse(v>m,1,0))
}

df <- airquality[,c(1,3,4)]  # Ozone, Wind, Temp
df <- df[complete.cases(df),]  # omit rows with NAs

# turn quant variables into factors
df$Ozone <- f(df$Ozone)
df$Wind <- f(df$Wind)
df$Temp <- f(df$Temp)

# build the net
bn1 <- hc(df)
plot(bn1)

# find the conditional probabilities
fit_air <- bn.fit(bn1, data=df)
fit_air
```

The output is shown below:

```
  Bayesian network parameters
  Parameters of node Ozone (multinomial distribution)
Conditional probability table:
         0         1
0.6206897 0.3793103

  Parameters of node Wind (multinomial distribution)
Conditional probability table:

    Ozone
Wind         0         1
   0 0.3611111 0.8181818
   1 0.6388889 0.1818182

  Parameters of node Temp (multinomial distribution)
Conditional probability table:

    Ozone
Temp         0         1
   0 0.7222222 0.0000000
   1 0.2777778 1.0000000
```

14.1.1 Querying the Network

Once we have the conditional probabilities determined by the bn.fit() function, we can query the net. Two queries run at the console and the results are shown below.

```
> cpquery(fit_air, event=(Ozone==1), evidence=(Temp==1))
[1] 0.6833689
> cpquery(fit_air, event=(Temp==1), evidence=(Ozone==1))
[1] 1
```

The cpquery() function performs conditional probability queries on the network. Specifically, it estimates the conditional probability of an event given evidence, and returns this probability. In the first query above we wanted to know P(Ozone=1|Temp=1), in other words, what is the probability that Ozone is high given that Temp is high. The value was 0.68. The second query asked for P(Temp=1|Ozone=1) which was 1.0.

14.2 Bayesian Net Semantics

A Bayesian network is a directed acyclic graph (DAG). In order to discuss properties of the network we first review some terminology from graph theory.

14.2.1 Review of Graph Terminology

A graph is defined by $G = (V, A)$ where V is the set of nodes or vertices and A is the set if arcs, links, or edges that connect the notes. An arc $a = (u, v)$ connects nodes u and v. If the connection is undirected, the order of the vertices does not matter. If the connection (u, v) is directed, the head will be u and the tail will be v. In Bayesian networks, all of the arcs will be directed. In a directed graph, if there is a path from v_i to v_j then v_i is an ancestor of v_j, which is a descendant. The direct ancestor is called a parent. The direct descendant is a child. Graphs can be either cyclic or acylic. Bayesian networks are acyclic, they can have no cycles.

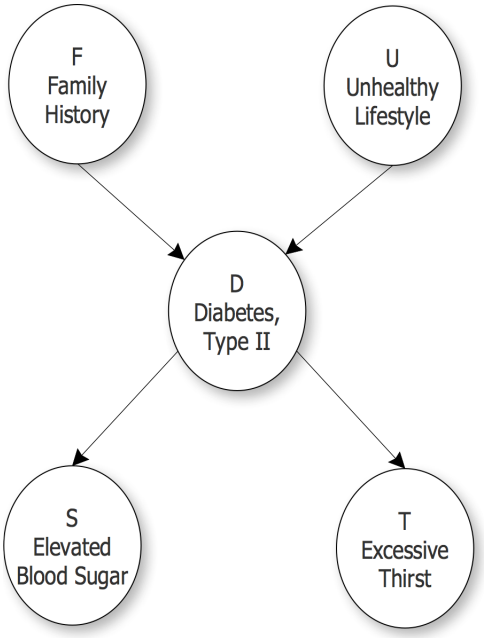


Figure 14.2: Bayesian Network

In Figure 14.2 we have a directed, acyclic graph, that provides an overly simple model of diabetes. We have 5 random variables: (1) F - a Yes/No variable for family history of diabetes, (2) U - a Yes/No variable for an unhealthy lifestyle, (3) D - Yes/No for a diagnosis of diabetes, (4) S - Yes/No for elevated blood sugar, and (5) T - Yes/No for excessive thirst.

14.2.2 Graph Structure

If two variables are independent, there will be no arc connecting them. Conditional independence is specified by the directed separation criterion (d-separation). The intuition behind d-separation can be seen in Figure 14.2. There is no direct path from F to S that does not go through D. D is said to d-separate F and S. Therefore, F is independent of S|D. This is denoted by: $F \perp\!\!\!\perp S|D$.

Directly following from the idea of d-separation is the Markov property of Bayesian networks. The Markov property for a Bayes Net states that there are no direct dependencies in the graph that are not shown explicitly. For example, the unhealthy lifestyle cannot affect

frequent thirst except through diabetes. Again, keep in mind this is an oversimplified example and not necessarily consistent with medical research.

The structure of a Bayesian network implies that the value of a variable is conditioned only on its parent nodes:

$$P(x_1, x_2, ..., x_n) = \prod_i P((x_i|Parents(X_i)) \tag{14.1}$$

In the case of the sample diabetes graph, P(Thirst|Diabetes, Unhealthy) = P(Thirst|Diabetes). This indicates a conditional independence of variables.

14.2.3 Reasoning with Graphs

The DAG of the network factorizes the global probability distribution into a local probability distribution for each variable. The connections provide means of reasoning about the variables. Figure 14.2 illustrates four types of reasoning we can do with Bayesian networks:

- diagnostic - given the evidence of excessive thirst, the cause is likely diabetes; notice this traces the graph in a bottom-to-top direction
- predictive - given diabetes, it is likely that a person experiences excessive thirst; notice this traces the graph in a top-to-bottom direction
- intercausal - given a family history leading to diabetes, this may explain away an unhealthy lifestyle as a factor
- combined - given a family history leading to diabetes, this in turn predicts excessive thirst

14.3 The Algorithm

The algorithm used in the code sample above, hc(), is a hill climbing algorithm. Hill climbing has widespread application in AI. In simple hill climbing for Bayesian networks, we start with an empty graph. Each variable in the data is evaluated by a score function that quantifies how well the network with the added node would fit the data. The search through the variables is greedy, adding the variables based on the highest score. Metrics vary, including a posteriori probability, or Bayes Information Criterion (BIC). There are many variations of hill climbing in AI and machine learning.

Another algorithm available in bnlearn() is TABU search, a constraint-based greedy search. The tabu search performs hill climbing until it finds a local optimum. It then searches through the next best variables that it has not visited recently, i.e., that are not on the tabu list.

14.4 Example: Coronary Data

As another example of a Bayes' net, we look at the coronary data set in R. The data specifies risk factors for coronary thrombosis for men. The data set has 1841 observations and 6 variables, all of which are binary factors:

- Smoking - yes or no
- M. Work - yes or no for strenuous mental work
- P. Work - yes or no for strenuous physical work
- Pressure - <140 or >140 systolic blood pressure
- Proteins - <3 or >3 ratio of beta and alpha lipoproteins
- Family - neg or pos for patient's indication of family history

Code 14.4.1 — Bayesian Network. Coronary Data.

```
library(bnlearn)
bn_coronary <- hc(coronary)
plot(bn_coronary)
```

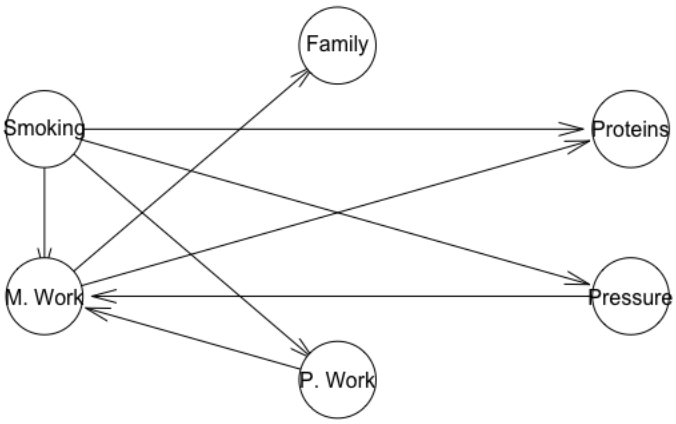


Figure 14.3: Bayesian Network for Coronary

Figure 14.3 shows the network for the coronary data. There are some links that make sense, such as Proteins being affected by smoking and mentally stressful work. There are other links that don't seem to make sense such as: Family being a child of mentally challenging work. We can list all links with "bn_coronary$arcs" at the console, and we see that the link we want to delete is the 7th one. We can remove this link as shown below.

```
bn_coronary$arcs <- bn_coronary$arcs[-c(7),]
```

Now if we replot, we see that the family node is still there but is not connected to any other node. Now we are ready to do some conditional probability queries.

```
> cpquery(fittedbn, event = (Pressure==">140"),
    evidence = ( Proteins=="<3" ) )
[1] 0.4310942
```

14.5 Summary

The term Bayesian Network was coined by Judea Pearl in 1985 to emphasize the Bayesian conditioning and the method for updating information. Judea Pearl was the 2011 winner of the ACM Turing Award.

Bayesian networks provide a graphical and highly interpretable representation of conditional probabilities in a data set. The examples in this chapter were for qualitative data but the bnlearn package can also handle quantitative data with a custom.fit() function rather than the bn.fit() function. Another package for Bayesian networks in R is DEAL, which provides several methods for using discrete or continuous variables.

14.5.1 Next-Level Learning

- Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Judea Pearl. Morgan Kaufmann. 1988.
- Bayesian Networks with Examples in R. M. Scutari and J.B. Denis. Chapman & HallCRC. 2014.
- Bayesian Networks in R with Applications in System Biology. Nagarajan, M. Scutari and S. Lebre. Springer. 2013.
- Article by M. Scutari, the author of bnlearn: `https://arxiv.org/pdf/0908.3817.pdf`

15. Markov Models

15.1 Overview

A hidden Markov Model, or HMM, is a probabilistic model often used to model sequential events such as temporal patterns, predictive text, part-of-speech tagging, and much more. First we discuss the simpler case of a Markov Model where we can see all the states, then move on to discuss the hidden Markov Model for unseen states which have observable results.

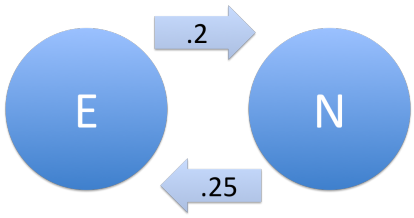


Figure 15.1: Markov Model for Two States: Exercise or Not

15.2 Markov Model

A Markov Model is a model for stochastic (random) processes. The model consists of a set of states and transition probabilities between states. The Markov assumption is that the probabilities for transition depend only on the previous state, not the entire string of preceeding states. Let's consider a simple system with 2 states modeling the likelihood that a person will exercise today, and is based on the observation a person made that if they exercised the

day before they tend to exercise today, with .8 probability, but if something interferred with exercising the day before they tend to not exercise today either, with .75 probability. State E represents the exercising state and state N represents the not exercising state.

As we see in the Figure above, a person in state E stays in state E with .8 probability and moves to state N with .2 probability. Likewise a person in state N stays in state N with probability .75 and moves to state E with probability .25. Below we see some R code representing the initial state of having exercised 5 of 30 days, and the transition matrix.

Code 15.2.1 — Initial State. Representing Exercise or Not.

```
# build the transition matrix for the model
transMatrix <- matrix(c(.8, .25, .2, .75), nrow=2)
transMatrix  # output matrix
     [,1] [,2]
[1,] 0.80 0.20
[2,] 0.25 0.75
# represent the initial state in the exercise matrix
exercise <- matrix(c(5/30, 25/30), nrow=2)
          [,1]
[1,] 0.1666667
[2,] 0.8333333
```

Initially, the person had a poor exercise pattern, exercising 5 days out of 30, about 17% of the time. What happens after 6 iterations?

Code 15.2.2 — Markov Process. Six iterations.

```
for (i in 1:6){
  exercise <- transMatrix %*% exercise
  print(paste("exercise at i=", i, ":",
        format(round(exercise[1,], 2))))
}
[1] "exercise at i= 1 : 0.34"
[1] "exercise at i= 2 : 0.44"
[1] "exercise at i= 3 : 0.49"
[1] "exercise at i= 4 : 0.52"
[1] "exercise at i= 5 : 0.54"
[1] "exercise at i= 6 : 0.54"
          [,1]
[1,] 0.5447909
[2,] 0.4552091
```

We see above that E has higher and higher probability as the process iterates. This model stabilizes after only 6 iterations, so even if we iterate 5000 times we end up at about the same place: .555 and .444. It changes slightly at each iteration but stabilizes at these values.

15.3 Hidden Markov Model

The limitation of the Markov Model is that it only encodes what we know. There may be hidden, or latent, variables that influence our model. How can we discover these? Through hidden Markov models, HMMs.

Figure 15.2 shows the basics of a hidden Markov model. We only see the observed data X. We assume there is some latent variable Z that manifests X. Notice there are transitions from Z state to Z state horizontally. We do not know what the transition probabilities are, but a hidden Markov model can help us discover patterns that could have generated our data.

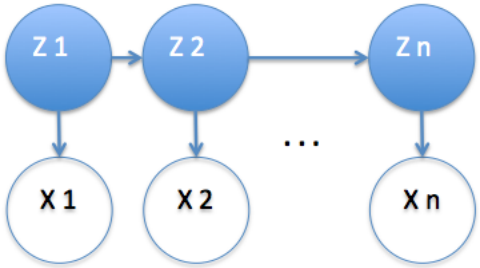


Figure 15.2: Hidden Markov Model

15.4 HMM in R

There are several packages that deal with HMM in R. We are going to look at `depmixS4`[1], a package by Visser and Speekenbrink. The package can generate mixture models including Markov models, hidden Markov models, and other mixture models. The documentation has several usage example. Below, we are going to continue our exercise model and see if we can detect a pattern in the data. First we load the depmixS4 package. Next we generate our observations. What we have are 7 transitions, one for each day of the week, and then we replicate that 10 times for 10 weeks. Interpreting the days in order, it seems that this person exercised (state 2) on Sunday and Saturday but not during the week.

Code 15.4.1 — HMM Fit the Model. Exercise Observations.

```
library(depmixS4)
#set of states
states <- c(2, 1) # E and NotE
n <- 140 # number of transitions (7 days, 10 weeks)
obs <-
  rep(c(c(2,2),c(2,1),c(1,1),c(1,1),c(1,1),c(1,1),c(1,2)),10)
```

Now that we have our observations, the Xs in Figure 15.2, we can let the algorithm find the transition probabilities. This is a 2-step process in this package. Step 1 creates the model

[1] https://cran.r-project.org/web/packages/depmixS4/vignettes/depmixS4.pdf

and Step 2 fits the model. Notice we also set a seed.

Code 15.4.2 — HMM Fit the Model. Exercise Observations.

```
# Start the HMM
set.seed(1234)
# 1. create the model
mod <- depmix(response = obs ~ 1, data=data.frame(obs), nstates=2)
# 2. fit the model
f <- fit(mod)
summary(f)
Initial state probabilties model
pr1 pr2
  1   0
Transition matrix
        toS1  toS2
fromS1 0.744 0.256
fromS2 0.100 0.900
```

We output the summary() above of the fitted model and displayed a portion of the output in the code box. Notice that the transition matrix was learned from the data. Now let's plot our results. First we extract the estimates from the fitted model. Then plot the actual observations over the estimates. Notice that the spikes of exercising (weekends) in the observations was matched by the estimates.

Code 15.4.3 — HMM continued. Extract Estimates and Plot.

```
estimates <- posterior(f)  # get the estimated state for each day
par(mfrow=c(2,1))
plot(1:n, obs, type='l', main='Observations, X')
plot(1:n, estimates[,2], type='l', main='Estimate')
```

15.5 Metrics

If we type the model name at the console we see the following:

```
> f
Convergence info: Log likelihood converged to within tol. (relative change)
'log Lik.' 4462.099 (df=7)
AIC:  -8910.198
BIC:  -8889.606
>
```

We are given information on the convergence, the log likelihood, AIC and BIC values. The AIC, Akaike information criterion, and BIC, Bayesian information criterion, are closely

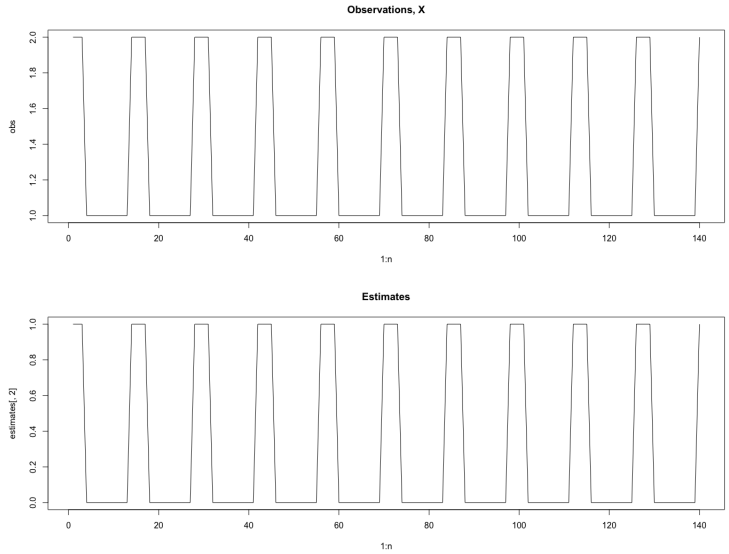


Figure 15.3: Observations and Estimates from HMM

related. AIC and BIC are typically used to select among models, here we only have one model. If we had more than one model and compared either AIC or BIC metrics, we would prefer the model with the lowest score. In, the standard formulas for AIC and BIC below, n is the number of observations, k is the number of parameters.

$$AIC = 2k - 2ln(\hat{L}) \qquad BIC = ln(n)k - 2ln(\hat{L}) \tag{15.1}$$

15.6 The Algorithm

In the hidden Markov model, we have n observations, $X = (x_1, x_2, ..., x_n)$. We do not know the parameters, θ that generated our observations but we can estimate them. First, let us consider the likelihood of observing X given these unknown parameters.

$$p(X|\theta) = \sum_{z} p(X, Z|\theta) \tag{15.2}$$

We can't really perform the summation directly in the above equation because we have N variables over K states. Our example above was binomial with K=2 but HMMs are often used for multinomial scenarios. We would need to calculate K^n terms, so this will grow exponentially with n. For this and other reasons, a direct solution is not feasible. Instead the EM, Expectation-Maximization algorithm is used to estimate a solution. In the application to HMM, the EM algorithm iteratively maximizes the expected joint log-likelihood of the parameters given the observations and states. That is the M step. The E step calculates expected values for the latent states given the observations and a set of initial states.

15.7 Another HMM in R

For this example we use the sp500 data set included in package depmixS4. First, let's load the package and look at the data. The column of interest is the 6th column, logret, the log ratio of the closing indices. For example the second row is 0.004 which can be calculated as `log(17.29/17.22)` at the console. We took the ratio of the index of the previous month to the current month, then the log. The range of this column is [-0.245, 0.15] and the mean (not shown) is 0.0058.

Code 15.7.1 — HMM. S& P500 data

```
library(depmixS4)
data(sp500)
head(sp500)
range(sp500[,6])

> head(sp500)
            Open  High   Low Close  Volume       logret
1950-02-28 17.22 17.22 17.22 17.22 1310000  0.009921295
1950-03-31 17.29 17.29 17.29 17.29 1880000  0.004056801
1950-04-28 17.96 17.96 17.96 17.96 2190000  0.038018763
1950-05-31 18.78 18.78 18.78 18.78 1530000  0.044645411
1950-06-30 17.69 17.69 17.69 17.69 2660000 -0.059792966
1950-07-31 17.84 17.84 17.84 17.84 1590000  0.008443619
> range(sp500[,6])
[1] -0.2454280  0.1510432
```

Now that we have our data loaded and understand the data in column 6, we can run the HMM algorithm and plot our results.

Code 15.7.2 — HMM continued. S& P500 data

```
# create the model, then fit
mod <- depmix(logret~1, nstates=2, data=sp500)
set.seed(1)
fmod <- fit(mod)

# plot
par(mfrow=c(3,1))
plot(posterior(fmod)[,1], type='l')
plot(posterior(fmod)[,2], type='l')
plot(sp500[,6], type='l')
```

The bottom graph is the actual column 6 data. The top model predicts the volatility going down, the middle one going up. We can definitely detect volatile months in the sequence. The sp500 data starts in February 1950 and ends in January 2012.

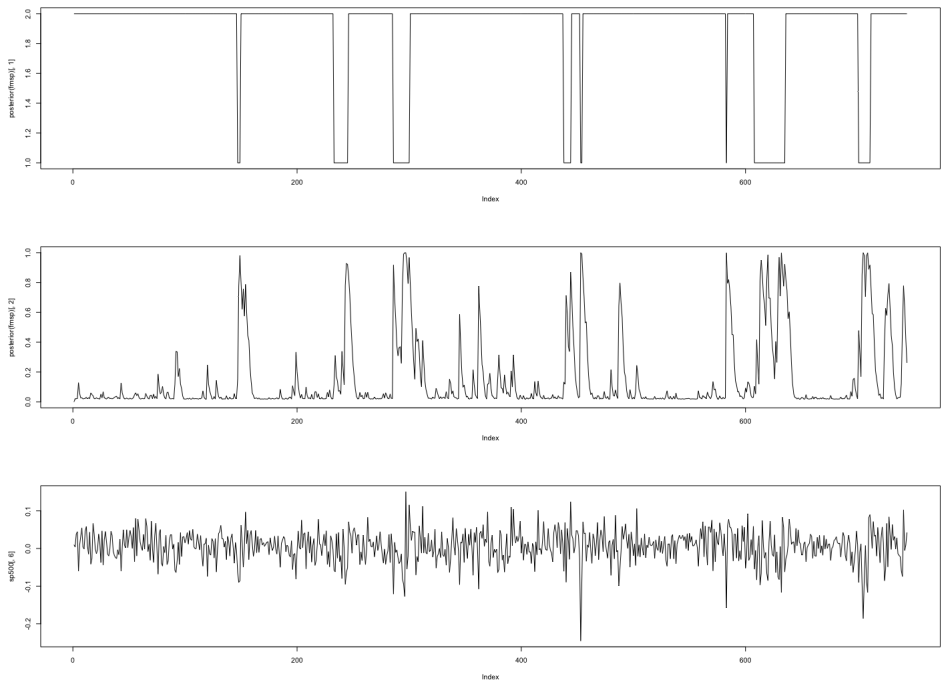


Figure 15.4: S&P 500 Data

15.8 Summary

In this chapter we explored hidden Markov models with package depmixS4. Another package worth investigating is seqHMM.[2] This package also outputs the transition probabilities in a graphical format.

The Markov of the Markov model is Andrey Markov, a Russian mathematician at the turn of the 20th Century. A Markov process assumes that future states depend only on the curent state, not states going back in time. A Markov chain describes a sequence of possible events in which the next state only depends on the current state. We saw a Markov chain in our first example of exercise or not exercise where we modeled these two states and the transition probabilities between them. HMMs are commonly used to model language, images, biological processes, virtually anything that is sequential in nature.

HMMs model hidden or latent variables that cause the observations. Variations of EM algorithms are often used to find optimal estimates for the latent variables and their transition probabilities. As we saw in the depmixS4 package, the algorithm discovers these probabilities and can output them for us.

A final thing to discuss is a Markov decision process. This is a Markov chain augmented with an action vector describing possible actions. This is often used in Reinforcement Learning, the subject of the next chapter.

[2]https://github.com/helske/seqHMM

15.8.1 Next-Level Learning

An excellent tutorial, *An Introduction to Hidden Markov Models and Bayesian Networks* is by Z. Ghahramani, and was published in the International Journal of Pattern Recognition and Artificial Intelligence[3].

[3]http://mlg.eng.cam.ac.uk/zoubin/papers/ijprai.pdf

16. Reinforcement Learning

Currently, Reinforcement Learning is a burgeoning field of AI, demonstrating success at improving self-driving cars, beating humans at the complex board game Go, optimizing data center energy usage, and much more. Reinforcement learning mimics how humans actually learn: a little trial and error, finding what works, and remembering. The reason that RL is taking off now is that it is being combined with deep learning. However in this chapter we discuss the traditional techniques of RL. OpenAI is providing a set of tools to widen participation in Deep RL. See this page: `https://spinningup.openai.com`

16.1 Overview

In previous chapters the focus was on learning from data for the purpose of either prediction or simply learning more about the data. In this chapter the focus shifts to learning how to act. We have a learner, an agent, who will learn to make decisions based on a utility function that keeps track of rewards earned for actions in different situations. The agent must learn to act under uncertainty, taking the action that most probably leads to a reward.

The core components of reinforcement learning are:

- an agent that interacts with the environment
- the environment, which is a set of predefined states, S
- a predefined set of actions, A, which the agent can take
- a set of rewards, R, that serve as reinforcement signals

The agent learns over many iterations of interaction with the environment. At each iteration, i, the agent observes the available states and selects an action. Based on the action chosen, the agent receives a numerical reward. After an action, a new set of states is available.

Learning, sometimes called Q-learning, relies upon remembering what was learned. The state-action function, $Q(s,a)$ defines the expected reward of each possible action for every state. A policy function $\pi^*(s,a)$, seeks to maximize the reward.

To jump-start learning, data in the form of sample state-action-reward sequences from which the agent learns can be fed into the learning algorithm. This *experience replay* can speed up convergence and help the agent learn faster.

16.2 The Markov Decision Process

In the last chapter we explored Markov Models which seem to have a lot in common with the reinforcement learning process described above. So what is the difference between RL and MDP? RL generally assumes there is some underlying Markov Decision Process, which it seeks to learn. In the MDP learning process, we know the states and possible actions at each time step, and the agent will receive a reward corresponding to the action chosen at the state. It is Markov in the sense that decisions consider only the current state, not previous states or actions. In Reinforcement Learning, the system may have to first learn the MDP. Another difference is that RL will try random actions in order to explore and learn new things beyond its current policy. So we can say that RL is an extension of the Markov decision process.

The Markov decision process is a 4-tuple <S, A, P, R> where S is the set of states, A is the set of actions, P() is a set of transition probabilities and $R(s,s',a)$ is the immediate reward for moving from state s to s' via action a. All of these are predefined. What we want to learn is the policy function $\pi()$ that maximizes a cumulative reward. More specifically, P() is:

$P(s,s',a) = Pr(s_{t+1} = s' | s_t = s, a_t - a)$

the probability that action a in state s at time t leads to state s' at time $t+1$.

Learning the policy $\pi(s)$ means learning the optimal choice of action at that state in order to maximize the long-term reward. The reward is often discounted by a factor, gamma, that ranges from 0 to 1. This serves to disincentive immediate reward in favor of long term reward. The reward over the long term (possibly infinite) can be expressed as:

$$\sum_{t=0}^{\infty} \gamma \, R(s_t, s_{t+1}, \pi(s_t)) \tag{16.1}$$

How is the policy learned? It could be learned by linear or dynamic programming, more commonly the latter which we discuss here. One approach is to set up two arrays, V for rewards, and π for the policy. Both arrays are indexed by the state:

$$\pi(s) := argmax_a \sum_{s'} P_a(s,s')[R_a(s,s') + \gamma \, V(s')] \tag{16.2}$$

$$V(s) := \sum_{s'} P_{\pi(s)}(s,s')[R_{\pi(s)}(s,s') + \gamma \, V(s')] \tag{16.3}$$

These definitions above are recursively updated through algorithms such as dynamic programming.

16.3 Reinforcement Learning

Reinforcement learning is closer to how humans learn than the Markov decision process because when we learn, we have to learn everything: what actions we can take, what states we may end up in, and what rewards our actions bring. No one gives their baby walking lessons, they just figure it out, through trial and error and encouragement from those around him. RL gives a computational foundations to let an agent learn from experience. RL is automated, goal-directed learning.

As in MDP, we have a policy which specifies how the agent should act at a given state, a value function that focuses on cumulative reward over time, a reward function, and some means of encoding the environment.

16.4 The ReinforcementLearning Package

We are going to explore the `ReinforcementLearning` package in R, available in CRAN. The following example utilizes the tictactoe data provided in the package that consists of over 400K game states. The agent must learn the optimal actions for each state of the board. The reward is 0 for tie, +1 for win, and -1 for lose. The following code will take a few minutes to run.

Code 16.4.1 — Reinforcement Learning. TicTacToe

```
library(ReinforcementLearning)
# Load dataset
data("tictactoe")

# Define reinforcement learning parameters
control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)

# Perform reinforcement learning
model <- ReinforcementLearning(tictactoe, s="State", a="Action",
    r = "Reward", s_new = "NextState", iter = 1, control = control)

# Print optimal policy
head(policy(model))
```

You can dig further into the model at the console by typing `model$` and pausing while the options pop up. For example, `model$Reward` lists the reward.

The three control parameters are for the behavior of the agent and all three range between 0 and 1 and have default values of 0.1.

- alpha - controls the learning rate; the lower the value the slower the learning; alpha=0 means nothing is learned
- gamma - discount factor determines the importance of future rewards; gamma=0 means that the agent only considers immediate rewards; gamma closer to 1 makes the agent work toward longer term rewards

- epsilon - exploration parameter, the probability that the agent will explore the environment through a random action;

There is another built-in sample experience, shown in the code block below.

Code 16.4.2 — Reinforcement Learning. Grid World

```
print(gridworldEnvironment)

# define states and actions
states <- c("s1", "s2", "s3", "s4")
actions <- c("up", "down", "left", "right")

# Generate 1000 iterations
sequences <- sampleExperience(N = 1000, env = gridworldEnvironment,
    states = states, actions = actions)

#Solve the problem
solver_rl <- ReinforcementLearning(sequences,
    s = "State", a = "Action",
    r = "Reward", s_new = "NextState")

#Getting the policy; this may be different for each run
solver_rl$Policy

#Getting the Reward; this may be different for each run
solver_rl$Reward
```

Printing the gridworldEnvironment shows the definition of the world. There are four states in a 2x2 arrangement with s1 and s2 over s2 and s3, respectively. The maximum reward is to get to s4. If you try to make a move that is not defined, you get minus 1 reward. The optimal policy, starting at s1, is to move down to s2, then right to s3, then up to s4. The policy at s4 will be different every time you run it unless you set a seed.

16.5 A Simple Python Example

Imagine a small environment seen in Figure 16.1. I credit Dr. Kathleen Swigger at The University of North Texas for the environment. The grid has dimensions 6x6. The agent has a start location and seeks to find a +1 location. Locations that are blank will have an initial reward of -0.01.

To find a path from start to the goal, initial states need to be represented:

- initialize a representation of the environment
- initialize q values based on the grid environment

Then try a number of runs:

$goal \leftarrow False$

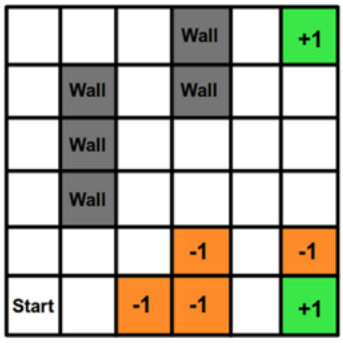


Figure 16.1: Sample Environment

while *NOT goal* **do**
  move ← *pick_move*
  make_move
  qvalues[*prior_location*] ← *update_reward*
end while

A very simple implementation is in the GitHub and summarized below. This minimal working of example is used just to explain one way of implementing Q learning.

16.5.1 Initialization

An environment is read in from a simple text file that gives the dimensions (6x6). A Python list of lists represents the environment. All locations are initialized to a reward of -0.01. The initial [row, col] location is set.

Next, q-values are initialized. This 3D list is of size 6x6x4 because at each location, 4 moves (up, down, left, right) can be chosen. The q-values are initialized by looking at the reward in the environment for each possible move. There are two locations with +1 as the reward, and 4 with -1 as the reward. All others are -0.01.

16.5.2 Pick a move

For a number of runs, the goal is set to False. Then an iterative process picks moves, makes moves, and updates the q-values.

A function to pick moves considers the 4 possibilities: up, down, left, right. Moves that would go beyond the grid, into a wall, or back to 'start' are not considered. Then the possible moves are randomly shuffled. If there is only one possible move, that move is returned. If there are multiple possible moves, they are searched for the move that brings the highest reward. That move will be returned.

16.5.3 Make a move

A function to make a move simply updates row or col, depending on if the move was up, down, left, or right. The new [row, col] location is returned.

16.5.4 Update previous q-value

Now that a move has been made and a reward received, the previous location's q-value can be updated:

```
reward =  qvalues[oldrow][oldcol][index]
qvalues[oldrow][oldcol][index] = reward + gamma * max_next_reward
```

The gamma was set to 0.8. A lower gamma in the range [0, 1] encourages the agent to pay attention to more immediate rewards. Higher gammas make the agent more willing to find a greater delayed reward. Most implementations also include an alpha parameter for a learning rate. The alpha parameter determines to what extent new information overrides what was previously learned. Alpha ranges from [0, 1] where 0 ignores new information and 1 ignores old information.

16.5.5 Results

With this simple implementation, the agent was able to reach the goal in decreasing numbers of turns in each learning episode. Because there is some random selection, each run of the program is different but the following output is typical for parameter settings allowing up to 50 learning episodes and a maximum total number of moves of 250:

```
Goal! Reached in 20 moves.
Goal! Reached in 24 moves.
Goal! Reached in 22 moves.
Goal! Reached in 24 moves.
Goal! Reached in 17 moves.
Goal! Reached in 28 moves.
Goal! Reached in 25 moves.
Goal! Reached in 16 moves.
Goal! Reached in 21 moves.
Goal! Reached in 19 moves.
Goal! Reached in 21 moves.
Goal! Reached in 23 moves.
Goal! Reached in 19 moves.
Goal! Reached in 30 moves.
Goal! Reached in 21 moves.
Goal! Reached in 19 moves.
Goal! Reached in 16 moves.
Goal! Reached in 10 moves.
Goal! Reached in 10 moves.
```

16.6 Summary

Reinforcement Learning has deep roots in AI using Markov foundations and techniques such as dynamic or linear programming. The most exciting trend in RL is Deep Reinforcement

Learning which is allowing systems to scale to problems unimaginable with traditional approaches. An excellent surved of deep RL from an IEEE Special issue can be found here: https://arxiv.org/pdf/1708.05866.pdf.

Full book on dynamic programming and RL is availble here: https://orbi.uliege.be/bitstream/2268/27963/1/book-FA-RL-DP.pdf

16.6.1 Next-Level Learning

A nice survey of the field, from historic roots to current trends is provided in this MIT Technology Review article: www.technologyreview.com.

DeepMind is a British AI company acquired by Google in 2014. Read about innovations at DeepMind: https://deepmind.com/blog/deep-reinforcement-learning/.

V Part V: Supplementary Materials

17. Data Wrangling

This chapter covers some helpful data manipulation techniques using both standard R and the tidyverse. The examples throughout this volume use data sets that have previously been curated by others. These include data sets built into R, data sets in R packages, and data sets available from sites such as `www.kaggle.com`. These ready-to-go data sets are used because our focus is on the algorithms, not the data wrangling. It is important to emphasize that clean data is not so easily obtained in the real world. The data gathering and cleaning phases in real-world machine learning projects can last months or even years. We can divide the data cleaning itself into two phases:

- data organization - compiling data into a form that can be read into an R data frame or other structure with functions such as read.csv() and read.table()
- data standardization - making sure that the values in the data frame make sense and are internally consistent

Most people think that data cleaning is not the most exciting phase of a project, but it is a chance to find problems in the data that could sink a project, and use that magnificent pattern-detection machine between your ears to discover insights.

17.1 Data Organization

Raw data can be gathered from many sources: data bases, unstructured data bases, xml files, or even scraped from the web. Programs to gather the data and put it in an R-compatible form can be written with R, but Python would actually be my first choice for text processing. In my GitHub I have a repo for Natural Language Processing that includes some sample code for text processing as well as web scraping.

17.1.1 Data Reorganization

Another form of data organization is when the data is in row, column format but not in the format you need for machine learning or data analysis. You may need to join or split rows or columns.

The following notebooks in the GitHub repo contain examples for data wrangling:

- 17_1-Data_Organization demonstrates how to manipulate columns and rows
- 17_2-Data_Standardization demonstrates techniques to clean up messy data
- 17_3-Data_Wrangling contains some of the code in this chapter

17.1.2 Reproducible Research

It is critically important to document each step of your data gathering and organization. This is important for your own sake in case you need to redo some steps. It is also important if and when you want to publish your results or report results to others. You will need to summarize how the data was collected and make detailed notes available for reviewers and fellow researchers. The most important reason to document is to create **reproducible research**. Anyone should be able to take your original data and follow your instructions to get the same results you reported.

17.2 Data Standardization

Once the data is gathered into a file in some regular format it can be read with read.csv() or read.table(). These functions read a file in table form and create a data frame. The assumption is that observations are organized into rows and columns represent attributes. You can use the help features of R to learn more about these read functions. We have used read.csv() like this:

```
df <- read.csv("data/myfile.csv")
```

There are numerous other arguments, here are a few of the most common ones:

- header=FALSE - the default case in which there is no header
- na.strings="NA" - use this to specify what strings are to be interpreted as NA
- stringsAsFactors = FALSE - this is the default since R 4.0; previously the default was TRUE

R automatically creates dummy variables for factors with more than 2 levels. You can check the coding with contrasts() or levels(). If R didn't interpret a column the way you intended, you can change this with the as.factor(), as.integer(), etc. functions as needed. It is never a good idea to assume that the data was read in the way you intended. Check your data with R functions such as str() and head(). Other important options with the read function include: changing row or column names, reading in only selected columns, some data cleaning regarding white space, and much more.

17.2.1 Dealing with NAs

Another problem is missing data, the NAs. We have discussed this before but repeat the code here for easy reference. First, you can check if NAs are going to be a problem by summing the number of NAs by column:

```
sapply(df, function(x) sum(is.na(x)==TRUE)
```

One option to get rid of NAs is to use the complete.cases() function. This will remove any row in which there is any NA in any column:

```
df <- df[complete.cases(df),]fix_NA
```

This may remove too much data. You should get rid of columns you don't care about before moving complete.cases() to limit how much data is lost. A less drastic option is to replace NAs with either the mean or median of the column. Here is a function to do this:

```
fix_NA <- function(x, mean_mode){
# sample call: df$x <- fix_NA(df$x, 1)
    if (mean_mode == 1) {  # use mean
        ifelse(!is.na(x), x, mean(x, na.rm=TRUE))
    } else {
        ifelse(!is.na(x), x, median(x, na.rm=TRUE))
    }
}
df$col <- fix_NA(df$col, 1)  # sample call
```

As we saw when performing classificaion on the *PimaIndiansDiabetes2* data set, if there are a large number of NAs in a given column, and we replace them with the mean or the median, we are essentially washing out the predictive power of that feature.

The dplr package has a fills() method that fills missing values in selected columns using the next or previous entry. This could be useful after values have been sorted on a related column.

Another approach would be to impute the missing value from neighbors. Set up a knn model for either classification or regression in which the target column is the column with the missing values and other columns are predictors. Then compare the predicted values to the values that are present in that column to see if the imputation was a good option.

17.2.2 Outliers

An *outlier* is a data point that exists far outside the range of the majority of the data. Outliers can be errors that were introduced at some point into the data gathering process. Determining whether data is truly an outlier or not takes some expertise beyond the scope of this book. The most important thing to keep in mind for the sake of reproducible research is to document every decision you make. If it is determined that a few values are truly outliers, they may be removed by replacing them with the mean or median values. Or, it may be best to remove that observation entirely by deleting the entire row. Again: document every decision made along with justifications that reflect your thinking at the time.

The boxplot() function can be used to plot the variable and detect outliers. In a boxplot, the circles at either the top or bottom of the plot may be outliers. Care must be taken to not

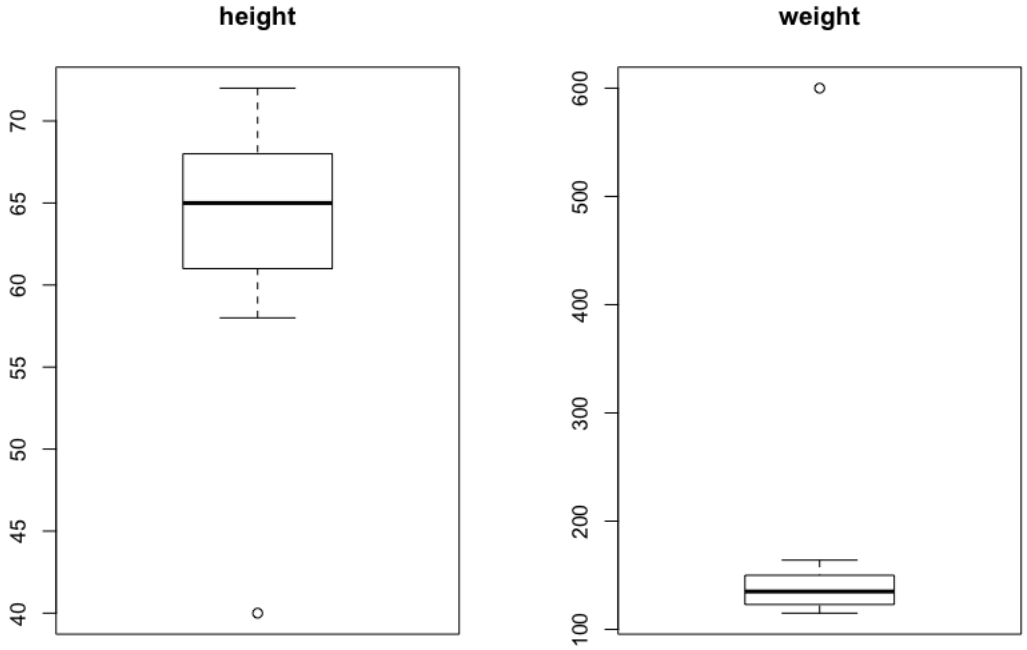


Figure 17.1: Outliers in Height and Weight Data

remove data just because it is inconvenient. In the plot above, there is one observation with a height well above the norm, and one observation with a weight well below the norm. To find which row is the suspected outlier, the which() function can be used, which.max(), and which.min() functions can be used, as shown in the sample code and output below.

Is it possible that a person could be 40 inches tall and weigh 120 pounds, or be 65 inches tall and weigh 600 pounds? Yes, these could be legitimate numbers. However, if the number of such outliers are very small, it will distort any attempt to learn something about average heights and weights of a population. For this reason, a researcher may decide to toss out these two examples, after thoroughly documenting their justification for doing so.

```
i <- which.min(df$height)
print(df[i,])
i <- which.max(df$weight)
print(df[i, ])
```

```
       height weight
16     40    120

       height weight
```

```
17     65    600
```

Sorting data can also help identify max and min values. The R order() function has the following parameters and default values as shown in this example which creates a new sorted data frame with the NAs last.

```
df_sorted <- order(df$colname, na.last=TRUE, decreasing=FALSE)
```

```
> head(df[order(df$height),], n=2)
   height weight
16     40    120
1      58    115
> head(df[order(df$weight, decreasing=TRUE),], n=2)
   height weight
17     65    600
15     72    164
```

Notebook 17_2-Data_Standardization in the GitHub contains several interesting examples of data wrangling to standardize the data.

17.3 Time Data in Base R

In base R, dates are stored internally as the number of days since January 1, 1970. Dates prior to this will be negative numbers. We can use the as.Date() function to convert a character date into an internally stored date. Interestingly, base R doesn't have a native time class. If you need one, look at package `hms`.

Here are a few examples of processing dates and times in base R. Note the built-in as.Date() and Sys.Date() functions. We can do basic arithmetic on days as shown below, and use the difftime() function with units of secs, mins, hours, days, or weeks.

Code 17.3.1 — Base R. Times and Dates.

```
# processing dates
hire_date <- as.Date("2016-09-01")
days_employed <- Sys.Date() - hire_date
print(days_employed)  # Time difference of 623 days

# processing time
birth_date <- as.Date("1989-04-18")
difftime(Sys.Date(), birth_date, units="secs")
# Time difference of 917654400 secs
```

17.3.1 Lubridate

The lubridate package contains functions to simplify time and date processing. Lubridate is part of the tidyverse.

Code 17.3.2 — Lubridate. Basics.

```
library(lubridate)
today()   # example: "2018-05-17"
now()     # example: "2018-05-17 15:51:37 CDT"
```

Next we show a couple of things you can do with lubridate and a data set. The airquality data set has a month and a day column. The year is 1973 as stated in the description of the data set. We pasted these together and used the ymd() function to create a date column which we added to the data frame.

Code 17.3.3 — Lubridate. Airquality data.

```
df <- airquality[]
df$date <- ymd(paste("1973",airquality$Month,airquality$Day))
print(range(df$date))
# "1973-05-01" "1973-09-30"
df$date[nrow(df)] - df$date[1]
# Time difference of 152 days
```

17.4 Text Data

There are various ways to handle text data, and different packages. In this section, we will look at two packages: tm and RTextTools that can be used for text processing. To be honest, R is not my first choice for machine learning with text data. See my GitHub for my Natural Language Processing materials if you are interested in learning on text. In this section we will look at two R packages for text.

17.4.1 Text Mining Packages tm

The text mining package, tm, uses vocabulary from the field of natural language processing, so we need to discuss that as we go. In NLP, a *corpus* is a body of text. It can be a set of documents, a set of text messages, any set of text examples. The online notebook gives an example of creating a corpus from an Amazon revew data set using the Corpus() function. After the corpus is created, some data cleaning is done: putting all text in lowercase, removing numbers and punctuation, stripping white space, and removing stopwords which are common function words that don't carry content like *the, an, this, it, in*, etc. Then a document-term matrix is created. This is a matrix in which each row represents a document in the corpus and each column represents a unique token (word) in the corpus. The cells hold word counts. Cell x, y holds the count of word y in document x. In the online notebook we see that the corpus contains about 4K documents and over 21K unique words. The document term matrix will be

sparse: most cells are 0, indicating that the majority of words do not occur in the document. Inspecting a portion of the document term matrix reveals this sparseness:

```
Docs bought send someone spent term thing unless want worst written
  50      0    0       0     0    0     0      0    0     0       0
  51      0    1       0     0    0     0      0    0     0       0
  52      0    2       0     0    0     0      0    0     0       0
  53      0    1       0     0    0     0      0    0     0       0
  54      0    0       0     0    0     0      0    0     0       0
  55      0    0       0     0    0     0      0    0     0       0
```

An alternative representation is to create a one-hot matrix. This involves replacing counts with 1 so that we have a binary present/not-present encoding of words in matrices.

As you can see in the online notebook, the reviews were divided into train and test sets and converted to one-hot matrix representations. The train matrix was used to train a naive Bayes algorithm and the test set was used to generate predictions. The naive Bayes algorithm had to learn probabilities for each word in the corpus. Below we show a couple of words. Most of the words had this kind of distribution, in that they provide as much evidence for a low rating as they do for a high rating. This approach was not sophisticated enough to learn anything from the words in the reviews, and is best suited to problems in which classes are more easily separable.

```
> nb1$tables$worst
                     worst
factor(train_labels)           0           1
                   1 0.9976470588 0.0023529412
                   2 0.9992877493 0.0007122507

> nb1$tables$best
                     best
factor(train_labels)          0          1
                   1 0.998235294 0.001764706
                   2 0.997863248 0.002136752
```

The tm package has some unique features that are interesting to experiment with. For example you can create a word cloud with selected words where the size of the word is based on the frequency of the word in the corpus.

17.4.2 RTextTools

Another online notebook provides examples of working with package RTextTools. This package was designed for non-technical people to do text analysis. Therefore it is simple to use but gives you less freedom to fine-tune the process. In the online notebook, the same

Amazon reviews data is read in and a document-term matrix is created as in the previous section. The RTextTools package uses the tm package for this. Then a container is created. This container holds the train and test split as well as the labels and will be fed into the machine learning algorithms. The algorithms included in the package are svm, glmnet, maxent, slda, boosting, bagging, rf, nnet, and tree. In the online notebook we used svm, maxent and glmnet. After classifying on the test data, the summary analytics can be printed as shown below:

```
ENSEMBLE SUMMARY
       n-ENSEMBLE COVERAGE n-ENSEMBLE RECALL
n >= 1                1.00              0.83
n >= 2                1.00              0.83
n >= 3                0.78              0.89

ALGORITHM PERFORMANCE
       SVM_PRECISION           SVM_RECALL           SVM_FSCORE
               0.820                0.815                0.815
    GLMNET_PRECISION        GLMNET_RECALL        GLMNET_FSCORE
               0.815                0.810                0.810
MAXENTROPY_PRECISION    MAXENTROPY_RECALL    MAXENTROPY_FSCORE
               0.805                0.805                0.800
```

For more about machine learning on text data, see my book *Exploring NLP with Python:* https://www.amazon.com/dp/B08P8QKDZK

18. Big Data with R

Big data is often characterized by the 7 Vs: volume, velocity, variety, veracity, value, visualization, vulnerability. Actually, the lists such as these vary from 4 to 10 or more, but you get the idea. This section provides a brief introduction to options for dealing with big data in R, along with resources to point you towards further information.

18.1 Memory, Data and R

On Mac or *nix, R will use as much memory as it needs, given what is available. On Windows, you can check the limit with `memory.limit()` and change the size with `memory.size()`. This is important because by default, R loads all data into virtual memory. The amounts available depend on your hardware and operating system. R uses two separate memory areas. Fixed-size objects called **Ncells** are stored in main memory and variable-sized objects called **Vcells** are stored on the heap. If you type `gc()` at the console, the garbage collection function will tell you how many cells you have of each type, with memory usage, and the garbage collection trigger threshold. In R, garbage collection is automatic but you can request it with gc() when you want, for example if you just removed a large object from memory with rm().

Here is a summary of some useful memory management console commands:

- ls() – list everything in memory
- rm(x) – remove x from memory
- rm(list=ls()) – remove everything from memory; caution!
- gc() – run the garbage collection function
- object.size(x) – find out how big x is
- memory.size() – find out how much memory you are using

- memory.limit() – find out the memory limit

Oddly, when I tried the last two commands on a 2023 iMac, I got the result of Infinite:

```
> memory.limit()
[1] Inf
> memory.size()
[1] Inf
```

Despite how you change your settings, or what your computer tells you, a rule of thumb is as follows. Data up to one million rows can be handled in standard R. Data sets from one million to a billion rows can be processed in R with some effort. Data over a billion rows will need to be handled with map reduce algorithms, combining R and Hadoop or some other platform. If the data you want to work with is too large, you can try the following:

- Make the data smaller. Try working with a subset of the data until you have your code written, then expand to all the data. Hopefully from the source of the data you can get a subset, especially if you can export a portion with SQL or other software. See the next chapter about sampling.
- Get more power. Increase your RAM, get a new computer, or try cloud computing services.
- Try MapReduce with package mapReduce.
- Try BigR with Amazon AWS or RStudio in the cloud.

18.1.1 Limit Data Size

You can limit the amount of data you read in by limiting the number of rows or limiting the number of columns. Limiting the number of rows can be done with various subsetting techniques or random selection from the full data, then remove the full data from memory. When reading in data with functions such as read.csv() you can use the `colClasses` argument to prevent reading in columns you don't want. The format is simple:

```
df <- read.csv("biggie.csv", colClasses = c("Name","City","Zip"))
```

18.2 Subset Data Base with dplyr

Let's say you are hired as a summer data science intern. You are excited to be working with a real data scientist on some big data. One week into the internship, your data scientist mentor is called away to another project in another city and tells you as she walks out the door, "Just see what you can get done while I'm gone." Oh my.

Before panic mode sets in, take a deep breath. You are working in an R environment and last week your supervisor asked you to go through the Big R materials at `https://www.rstudio.com/resources/webinars/working-with-big-data-in-r/`. You learned that based on RStudio's discussion with clients, there are some common steps that big data exploration projects commonly go through:

- Clarify - become familiar with the data

- Develop - create a working model
- Productize - automate the work so that it can be updated and reproduced by others
- Publish - share with others

So, starting at the beginning, you want to learn more about the data. But it's really big and stored on a server. It won't fit on your computer. Why not subset it for exploration purposes, and you can scale up later? This is actually a common approach to big data.

There are many R packages that function as an API to data stores, such as dplyr, DBI, RHadoop, SparkR, and more. Typically companies use RStudio Server Pro. A nice feature of R is that you can use dyplr on the front end for data wrangling and the back in can plug into many different formats such as databases through a SQL interface. There are many functions that can be used to connect to the database such as src_postgres, src_sqlite, src_mysql, and so forth, that use the language of your data base.

The tutorial referenced above gives example code for building SQL code to select data from a remote data base, then subset by 1% to give a sufficient amount of data to build a working model. How to you eat an elephant? One bite at a time. Likewise you will slice a portion of the data to go through steps one and two to learn about the data and build a working model to impress your supervisor when she gets back to town.

18.3 The ff Package

The `ff` package provides a way to have your big data on disk while processing it a section at a time in RAM. To use the package, you must `install.packages("ffbase")` and this will install everything you need. The ff package effectively swaps pages from the disk file into RAM in the background so that it will appear to your script that the file is all in RAM. We are going to discuss a few examples below. To learn more, read the full documentation for the package, available here: `https://cran.r-project.org/web/packages/ff/ff.pdf`.

18.3.1 Read in the data

The code below shows that we have to load the ff package first, then we can read in our csv. This csv file was downloaded from Kaggle, from the PUBG video game match data. Using the arg VERBOSE=TRUE lets you know how it's progressing on the file. Otherwise you might think it hung up on a large file. This file took about 10 minutes to load, 667 seconds to be more precise, on a 2013 MacBook with 8G RAM. The file on disk is 1.74 GB on disk, yet the R environment pane tells me that d is a large ffdf object of 12 elements and size 395.8 Mb. This indicates the fact that it is working on a chunk at a time. The R notebook is not available on the github because this data set belongs to Kaggle, it is not in the public domain.

Code 18.3.1 — ff. Read in data

```
require(ffbase)
d <- read.table.ffdf(file="kill_match_stats_final_0.csv",
   FUN="read.csv", header=TRUE, VERBOSE=TRUE, na.strings="")
```

Now that we have object d loaded, we can do a few data exploration tasks. First, we ask R

what kind of object it is, and it tells us it is a "ffdf" object. The dimensions are 11,653,619 rows and 12 columns, which would definitely not fit in RAM on the computer we used. Notice that you can use the same R functions we are used to such as dim(), but in the case of str() we chose to subset it.

Code 18.3.2 — ff. Data Exploration

```
class(d)   # ffdf
dim(d)     # 11,653,619 x 12
str(d[1:10,])
```

```
'data.frame': 10 obs. of  12 variables:
 $ killed_by        : Factor w/ 57 levels "AKM","AWM","Bluezone",..
 $ killer_name      : Factor w/ 2394678 levels "#unknown","18cmGirl",..
 $ killer_placement : num  5 31 43 9 9 26 12 27 40 25
 $ killer_position_x: num  657725 93091 366921 472014 473358 ...
 $ killer_position_y: num  146275 722236 421624 313275 318340 ...
 $ map              : Factor w/ 2 levels "ERANGEL","MIRAMAR": 2 2 2
 $ match_id         : Factor w/ 128728 levels ,...
 $ time             : int  823 194 103 1018 1018 123 886 137 89 117
 $ victim_name      : Factor w/ 3924033 levels "#unknown","0219i",...
 $ victim_placement : num  5 33 46 13 13 47 15 38 47 43
 $ victim_position_x: num  657725 92239 367304 476646 473588 ...
 $ victim_position_y: num  146275 723375 421216 316758 318419 ...
```

Here are a few more data exploration examples, with the results shown in the comments. First, we asked how many rows indicate a location of MIRAMAR instead of the other location ERANGEL. Then we discovered than the range of time values was 28 to 2374. Finally, that we have about 2 million unique killers out of 11 million rows.

We can also create graphs from the full data set with a little help from the cumulative sum function.

Code 18.3.3 — ff. Plots

```
par(mfrow=c(2,1))
hist(cumsum.ff(d$victim_placement, na.rm=TRUE)[1:8], main="")
hist(cumsum.ff(d$killer_placement, na.rm=TRUE)[1:8], main="")
```

Finally, we subset the data to a smaller object so that we can run some algorithms on the smaller data before working in all the data.

Code 18.3.4 — ff. Subset

```
i <- sample(nrow(d), 1000, replace=FALSE)
small_d <- d[i,]
dim(small_d)  # 1000 x 12
```

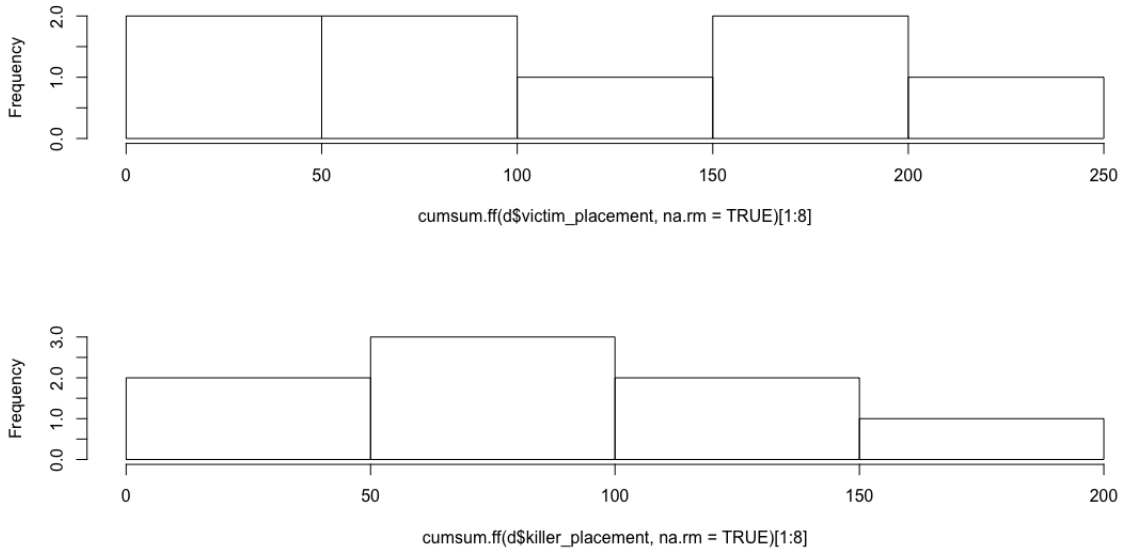


Figure 18.1: Residuals Plot for lm3

```
head(small_d, n=2)
```

	killed_by <fctr>	killer_name <fctr>	killer_placement <dbl>	killer_position_x <dbl>	killer_position_y <dbl>	map <fctr>
5312002	S1897	SkipToMyLouuu_	20	358210.1	261712.9	MIRAMAR
7298565	Down and Out	wosaaas	39	686839.0	460346.1	ERANGEL

2 rows | 1-7 of 12 columns

Figure 18.2: Residuals Plot for lm3

Check out these resources for more on big data with R:

- Dhafer Malouche has an extensive tutorial on the **ff** libraries: `https://malouche.github.io/bigdata2018/largedata.html`
- Jeffrey Breen has this tutorial on big data and R: `https://github.com/jeffreybreen/tutorial-201209-TDWI-big-data`
- This link `https://www.r-bloggers.com/2018/12/an-introduction-to-h2o-using-r/` is an introduction to the H2O machine learning and big data platform which works with R
- Posit Cloud (`https://posit.cloud/`) can easily handle large data and they offer academic discounts
- Posit Cloud also has an interface to Apache Spark: `https://spark.posit.co/`

18.4 Amazon Cloud Services

If you have an Amazon account, you can go to Amazon web services `https://console.aws.amazon.com` and look at the options under Machine Learning. Not all of the tutorials are free and the costs of AWS services varies depending on what you are doing, so tread carefully.

One option for big data in the cloud in R is sparklyr, an R interface to Spark. Apache Spark is an open source platform for big data. Spark is fast and is designed to play nicely with R, SQL, Python, Java and Scala. Spark can run stand-alone or in the cloud, on Hadoop, Apache Mesos, Kubernetes and more, and can input diverse data sources. The R package `sparklyr` connects to Spark from R and provides a dplyr backend. One approach to big data with sparklyr would be to scale down big data to download on a machine for use in R. Another approach is to use Spark's distributed machine learning library if you want to keep your big data in the cloud. Amazon EMR is a big data service on AWS that provides Spark (as well as other big data applications). A tutorial for AWS Spark using R is available here: `https://aws.amazon.com/blogs/big-data/running-sparklyr-rstudios-r-interface-to-spark-on-amazon-emr/`. The code should look very familiar, there are just a few extra things to do in the AWS CLI to get set up. Before you get started you have to install the CLI on your computer so you can control your AWS from the terminal. You can find information on the CLI in Amazon.

In addition, RStudio maintains a page about Cloud Services that includes instructions for AWS: `https://tensorflow.rstudio.com/tools/cloud_server_gpu.html`

18.5 Google Cloud Services

Google calls their cloud service Google CloudML and it does have support for RStudio. RStudio maintains information about Google Cloud here: `https://tensorflow.rstudio.com/`

18.6 Big Data: A Sham?

In a recent article[1], Brian Millar observes that while the big players are collecting and keeping big data, they don't actually use that much of it. He quotes ex-Googler Seth Stephens-Davidowits as saying "At Google, major decisions are based on only a tiny sampling of their data. you don't always need a ton of data to find the right insights. You need the right data." The article goes on the discuss the merits of sampling big data and running analysis on that sample, rather than forcing oceans of data through the straw. Sampling is the topic of the next chapter. The article referenced here is definitely worth reading, as is the book he mentions by Stephens-Davidowits, *Everybody Lies: Big Data, New Data, and What the Internet Can Tell Us About Who We Really Are*.

[1] https://www.fastcodesign.com/90168426/big-data-is-a-sham

18.7 Sampling Big Data

The data sets used through the main portion of this volumne are small enough to fit into memory without using any of the special techniques described in this chapter. If you are tackling big data, an alternative to using the approaches described in the previous chapter is to use sampling. The technique of *sampling* is used in statistics to estimate characteristics of a large population by selecting a representative subset and determining characteristics of the subset. A thorough coverage of sampling is beyond the scope of this handbook. However, we can discuss a few key elements of sampling to give background information for those who will be working with data scientists and statisticians.

Figure 18.3: The Concept of Sampling

18.7.1 What is Sampling?

Sampling is a thoroughly researched topic in statistics that was used because data collection was expensive. For example, calling people for political surveys. With big data, the collection is already done, but how can we go about analyzing it? Again, sampling to the rescue. By cutting the data down to size, its analysis becomes easier. And by size we mean rows, the number of observations. There are also techniques for cutting down columns, such as dimensionality reduction techniques like PCA discussed in other chapters.

There are many, many approaches to sampling. Throughout the book, when we divided our data sets into train and test, we used random sampling. Each observation had an equal chance of being selected. Another common approach is to use stratified sampling, which involves dividing data into k buckets and making sure each bucket is represented in the sample. As a simple example, let's say we want the mean income in a data set. We can take the weighted mean of each of k buckets to estimate mean income of the population:

$$\frac{n_1\bar{X}_1 + n_2\bar{X}_2 + ... + n_k\bar{X}_k}{\sum_i^k N_i} \tag{18.1}$$

18.7.2 Sampling Example

An online notebook for this chapter demonstrates a sampling example using a kaggle data set on credit defaults. The data set has 30K observations and we included 11 variables. The distribution of our target column, default, has 22% of the examples with default = true. This data is not "big" by any stretch of the imagination but it demonstrates techniques that could be used on big data. We approach this data set with two primary quesions: (1) does sample size matter? and (2) does having an unbalanced data set matter?

First, we randomly divided the 30K examples into 80% train and 20% test. Running logistic regression on the full data set resulted in an accuracy of 80%. This is the "Full Data Set" row in Table 18.1, which shows accuracy, sensitivity and specificity. Recall from earlier in the book that sensitivity is the percentage of accurately classified positive examples (default=false) and that specificity is the percentage of accurately classified negative examles (default=true). For the second experiment, we randomly sampled 1000 observations from the 30,000 and divided this into an 80-20 split. This is shown in the second row of the table below. The sensitivity was about the same but there is a marked improvement in specificity and thus overall accuracy. The confusion matrix for this run shows that only 19% of the test cases were defaults compared to 23% of the test cases in the full data set example, which could partially account for the improvement in accuracy. For our third example we used the createDataPartition() function in package caret to make sure our test examples contain a percentage of default cases simlar to the entire data set. We got identical results as in the random sampling. Finally, we limited the number of default and no-default cases to 5000 each, trained the algorithm on this equally split data, and tested on the same test data as row 1. The results are a decrease in accuracy. Sensitivity dropped by 25% but specificity increased by 14%, resulting in an overall accuracy decrease of 9%.

Sample	**Accuracy**	**Sensitivity**	**Specificity**
Full Data Set	80%	97%	23%
1000 Data Set	84%	96%	36%
Full Stratified	80%	97%	23%
50-50	71%	74%	59%

Table 18.1: Results from Sampling Variations

A 2012 article by Crone and Finlay from the International Journal of Forecasting[2] can help us answer our earlier questions about sample size and balanced data. The authors note that expert opinion on sampling credit data is that 1500 instances of each class is sufficient and that biased data sets should be balanced. From our own experiment above we see that the larger data did not perform better than a smaller subset and that balancing the data set resulted in decreased accuracy. The authors experimented with various sample sizes on logistic regression, discriminant analysis, decision trees and neural networks on two different data sets. The authors found that more data than the recommended 1500 per class improved accuracy, but not for logistic regression which does not required large amounts of data for optimal performance. Regarding data balancing, they found that oversampling the minority class is preferred to undersampling the majority class across all algorithms. Again, they note that logistic regression is robust to the data distribution so such sampling would not be of benefit for that algorithm.

Another examination of balancing data is found in a blog post[3] by Nina Zumel of Win-

[2]https://doi.org/10.1016/j.ijforecast.2011.07.006

[3]http://www.win-vector.com/blog/2015/02/does-balancing-classes-improve-classifier-performance/

Vector LLC, a data science firm from San Francisco. Three algorithms were compared: regularized logistic regression, randomForest, and soft-margin SVM. Her results also showed that logistic regression performance was worse with a more balanced data set. SVM degraded only slightly and random forest improved but it was the lowest-performing algorithm across all samples of the data.

18.8 Current Practices

So what do data scientists and statisticians usually do with big data? A 2017 paper by Rojas et al.[4] gives insight into what data scientists are currently doing. A total of 22 data scientists working at companies like Google, Microsoft, etc. were surveyed. The survey results showed that the majority used random sampling, stratified sampling and sampling by hand. Although other sampling techniques may help data scientists gain more insight into data, they tend to not be used because the data scientists were not exposed to these ideas in their formal education and because of concerns that different sampling techniques might introduce bias. The hypothesis of the paper, confirmed by their results, was that data scientists could gain better insight into data if they used multiple sampling techniques rather than one. An experiment was conducted to probe what insights data scientists glean from 4 different sampling techniques:

- random sampling
- density sampling
- uncertainty sampling
- QBC query by committee

The results of the experiment indicate that data scientists will gain additional insights by using more than one sampling technique because they each have their strengths and weaknesses. Density sampling filters out outliers to enable seeing general trends more clearly. QBC looks at features near the separating boundaries of classes, thereby highlighting features that do not indicate general trends. Uncertainty sampling focuses on outliers and the features signifying outliers. Using these techniques in addition to the current practice of random sampling gives data scientists additional insights.

A blog post by Alex Gold outlines different strategies for working with Big Data in R, including sampling. Read his post here: `https://rviews.rstudio.com/2019/07/17/3-big-data-strategies-for-r/`

[4] http://www.rosenthalphd.com/papers/RosenthalRojas_LDAV17.pdf

19. Statistics Resources

This chapter is included to put much of the statistical discussion throughout the volume in one place for easier reference. This chapter will also point you to additional resources should you find yourself needing a deeper dive. Rather than your having to search for where to learn more about these topics, this chapter will point you in the right direction.

Classical statistics, often called frequentist statistics, is probably the statistics you learned in your Intro to Stats class. Classical statistics is based on the frequencies of random events in a long series of trials. From data collected through observation, inferences can be made, chiefly about entire populations based on samples. With classical statistics, you can compute statistical significance, which is critical in scientific publications. Classical statistics give us the foundation of null hypothesis testing. An alternative approach to statistics is Bayesian which we discuss briefly at the end of the chapter and point you to the perfect resource to learn more.

19.1 Data and metrics

Classical statistics deals with numeric and categorical data, which can be divided further:

- Numeric data
  - Continuous, such as real numbers that can take on any value in an interval
  - Discrete, such as integers or counts
- Categorical data
  - Binary, as in 0/1 or True/False
  - Factors, categorical data that can take on a set of values
  - Ordinal, categorical data that has an explicit ordering

In this volume we have worked with continuous numeric data, as well as binary and factor data. These are the types of data encountered in our data sets. R can handle any of the above types of data, and we will look at some examples below.

19.1.1 Estimates

Rather than give formulas in this discussion, the approach will be more intuitive. Some formulas appeared earlier in this book, and are replicated in the Github notebooks, but others can be found on the cheat sheet[1] from MIT that is included in the GitHub.

Measures of central tendency are often important insights into where individual data items tend to be located. In the R chapter we explored how to compute such statistics as mean (average), median (middle point), range (smallest to largest), and so forth. We also discussed methods to detect outliers, those values that seem different from most of the data. See the online notebook, Appendix C.

In addition, we explored estimates of variability such as variance, covariance, correlation.

R **Estimates** Measures of central tendency such as mean and median apply to a vector. They give us insight into where most of the data resides on a spectrum. Measures of variability such as variance and correlation compare two vectors.

Notebook 19-1 in the GitHub demonstrates:

- Graphs for initial data exploration of a qualitative or quantitative variable. See Notebooks 3-1, 3-2, and 4-2 for more examples.
- Measures of Central Tendency
- Measures of Variability
- N-1 and degrees of freedom
- Correlation
- Variance
- Estimates based on percentiles

19.2 Data Distributions

Data distributions were discussed in Chapter 8, Section 4. Notebook 19-2 revisits that discussion with the formulas and charts that were used to create the graphics in Chapter 8. The distributions reviewed in Notebook 19-2 include:

- Normal distribution aka Gaussian distribution
- Binomial distribution
- Multinomial distribution
- Dirichlet

[1] https://web.mit.edu/~csvoss/Public/usabo/stats_handout.pdf

19.3 Stats for Linear Model Evaluation

Notebook 19-3 in the GitHub reviews some statistics that cropped up in our discussion of Linear Regression, and which have general relevance in machine learning and data science:

- Random sampling and Bias
- Central Limit Theorem
- Confidence Intervals, t-values and p-values
- Compute a confidence interval
- Standard normal distribution and z scores
- Student's t distribution

19.4 Summary

One could spend an entire career exploring statistics and this pursuit could be quite lucrative. Companies have data and they need to learn from it. For me, I've chosen to merely dip my toe in the field when I needed to for specific projects. If you plan to be in the data science or machine learning fields, it helps to build a physical library of books that you can grab quickly rather than going down Internet rabbit holes. Start with introductory books and build your library over time. Here are some introductoriy books I've found helpful over the years.

- *Discovering Statistics using R* by Field, Miles, Field. This is a from-the-ground-up introduction to statistics and the R language.
- *R in Action* by Kabacoff. Introduction to data analytics and graphics in R.
- *Statistical Computing with R* by Rizzo. Covers basic of statistics and R along with advanced topics such as Monte Carlo methods and Markov chains.
- *Advanced R* by Hadley Wickham. A deep dive into how R works under the hood in order to build a foundation for writing R libraries.
- *Statistical Rethinking* by McElreath. A Bayesian approach to statistics using R.
- *Practical Statistics for Data Scientists* by Bruce, Bruce, Gedeck. A good reference for statistical techniques with code samples in R and Python.
- *The Seven Pillars of Statistical Wisdom* by Stigler. A delightful high-level tour of the history and application of statistics.

19.4.1 Next-Level Learning

There are a couple of areas of study that may be particularly useful to you as either a machine-learning practitioner or a data scientist: understanding A/B testing, and Bayesian statistics.

Experiments and Evaluation

Why do machine learning practitioners care about these kinds of scientific experiments? Generally, they don't, but social media platforms conduct A/B tests on various layouts and more of the platform to see which versions gets more clicks and longer eyeball times. Yes, we are the lab rats my friend.

We highly recommend an excellent tutorial on doing A/B testing in R for market analytics.[2].

[2] `https://www.r-bloggers.com/2023/12/introduction-to-a-b-testing-in-r-for-marketing-analytics/`

The tutorial includes a link to a one-hour workshop on YouTube that includes the R Bloggers notebook content and much more. The tutorial teaches real-world skills that can get you a job and/or clients.

Bayesian Statistics

The fundamental difference between classical (aka Frequentist) statistics and Bayesian statistics is that classical statistics assumes that underlying probabilities are fixed whereas Bayesian statistics assumes that probabilities can be updated based on new information. Bayesian statistics incorporates prior knowledge, and treats parameters as random variables with probability distributions. Bayesian statistics, no surprise, relies on Bayes' Theorem that we learned when exploring Naive Bayes, as well as Bayes' Nets.

An entire, and surprisingly delightful, introduction to Bayesian statistics is given in the book *Statistical Rethinking*, by Richard McElreath. An excellent summer project for students would be to go through this book which includes hands-on code interspersed with conceptual introductions.

20. Sharing Work

One of the most amazing things about the RStudio IDE is the ability to render your work to html or pdf for easy sharing with others. In this chapter we will look at two other RStudio innovations for sharing work: *Shiny* and *Quarto*. This chapter will just give you a taste of these applications, with pointers to where to learn more.

20.1 Shiny

To see a sample interactive Shiny app, open RStudio and install Shiny:

```
install.packages("shiny")
```

Then you can run the first example:

```
library(shiny)
runExample("01_hello")
```

This will pop up the app, which is just a histogram of a vector of waiting times for volcanic eruptions. The html-based page has a slider with which a user could adjust the number of bins displayed in the histogram. Several examples are shown on the documentation site (`https://shiny.posit.co/r/getstarted/shiny-basics/lesson1/`) to get a feel for how interactive these apps can be.

20.1.1 Structure of a Shiny app

So how did this magic happen? The Shiny app is contained in a script called **app.R** which is in its own directory, and can be run with a simple command such as: `runApp("nydir")`. The **app.R** file requires 3 components:

- a user interface object which controls the layout and appearance of the app
- a server function to give instructions to your computer how to build the app
- a call to the shinyApp() function which does the heavy lifting based on the ue and the server functions

Go to the Gallery page (`https://shiny.posit.co/r/gallery/`) to see some amazing demos of what Shiny could do. Many resources are available as well to help you build something amazing.

20.2 Quarto

RStudio has transitioned to *posit* with an expanded mission to support reproducible research. Quarto makes it easier to do RStudio kinds of magic with other languages in addition to R, such as Python and Julia. The RStudio IDE already supports creating and rendering Quarto documents. Recall that the R notebook files you created had extension Rmd. If you create a Quarto file it will have extension Qmd. There are extensions in VSCode to enable working both the R and Quarto. Extensions are also available for other platforms.

One of the main advantages of Quarto is the variety of output formats that can be created such as websites, books and blogs. Quarto is considered to be the next evolution of Rmarkdown by the posit team, as is largely back compatible with notebooks that have been created as we have done in this volume. The main difference will be in the YAML headings to facilitate the variety of output formats.

If I had to summarize why someone would move from Rmarkdown to Quarto I would put it this way: Quarto facilitates doing your work once (in multiple languages no less) using whatever IDE you prefer (RStudio, VSCode, Emacs) and by simply changing the YAML header (and maybe a few more things), you can render it into a variety of output formats.

20.3 Summary

Once a Shiny or Quarto work has been created, it can be shared on Shiny Servier or Posit Connect. See the documentation; `https://posit.co/products`

The posit team has also started a project called tidymodels which can be used for machine learning. At the current development level, it appears to be most appropriate for professionals who have not studied machine learning, but stay tuned for future development.