R for Statistical Learning

David Dalpiaz

2017-02-18

Contents

Introduction

Welcome to `R` for Statistical Learning!

0.1 About This Book

This book will serve as a supplement to An Introduction to Statistical Learning for STAT 430 - Basics of Statistical Learning at the University of Illinois at Urbana-Champaign.

Chapters will come in roughly three flavors:

- **Notes** that discuss mathematics in greater detail.
- **Tutorials** that illustrate the use of `R` for statistical learning.
- **Analyses** that show end-to-end analysis of a particular dataset.

This book is under active development. Chapters will be added as we move through the course. Often they will be more in the style of course notes than a fully narrative text.

When possible, it would be best to always access the text online to be sure you are using the most up-to-date version. Also, the html version provides additional features such as changing text size, font, and colors. If you are in need of a local copy, a **pdf version** is continuously maintained.

Since this book is under active development you may encounter errors ranging from typos, to broken code, to poorly explained topics. If you do, please let us know! Simply send an email and we will make the changes as soon as possible. (`dalpiaz2 AT illinois DOT edu`) Or, if you know RMarkdown and are familiar with GitHub, make a pull request and fix an issue yourself! This process is partially automated by the edit button in the top-left corner of the html version. If your suggestion or fix becomes part of the book, you will be added to the list at the end of this chapter. We'll also link to your GitHub account, or personal website upon request.

This text uses MathJax to render mathematical notation for the web. Occasionally, but rarely, a JavaScript error will prevent MathJax from rendering correctly. In this case, you will see the "code" instead of the expected mathematical equations. From experience, this is almost always fixed by simply refreshing the page. You'll also notice that if you right-click any equation you can obtain the MathML Code (for copying into Microsoft Word) or the TeX command used to generate the equation.

$$a^2 + b^2 = c^2$$

0.2 Conventions

`R` code will be typeset using a `monospace` font which is syntax highlighted.

```
a = 3
b = 4
sqrt(a ^ 2 + b ^ 2)
```

R output lines, which would appear in the console will begin with ##. They will generally not be syntax highlighted.

```
## [1] 5
```

Often the sybmol $\triangleq$ will be used to mean "is defined to be."

0.3 Acknowledgements

Your name could be here! Suggest an edit! Correct a typo!

- James Balamuta, Summer 2016 - ???
- Korawat Tanwisuth, Spring 2017
- Yiming Gao, Spring 2017

0.4 License

Figure 1: This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Chapter 1

Probability Review

We give a very brief review of some necessary probability concepts. As the treatment is less than complete, a list of references is given at the end of the chapter. For example, we ignore the usual recap of basic set theory and omit proofs and examples.

1.1 Probability Models

When discussing probability models, we speak of random **experiments** that produce one of a number of possible **outcomes**.

A **probability model** that describes the uncertainty of an experiment consists of two elements:

- The **sample space**, often denoted as Ω, which is a set that contains all possible outcomes.
- A **probability function** that assigns to an event A a nonnegative number, $P[A]$, that represents how likely it is that event A occurs as a result of the experiment.

We call $P[A]$ the **probability** of event A. An **event** A could be any subset of the sample space, not necessarily a single possible outcome. The probability law must follow a number of rules, which are the result of a set of axioms that we introduce now.

1.2 Probability Axioms

Given a sample space Ω for a particular experiment, the **probability function** associated with the experiment must satisfy the following axioms.

1. *Nonnegativity*: $P[A] \geq 0$ for any event $A \subset \Omega$.
2. *Normalization*: $P[\Omega] = 1$. That is, the probability of the entire space is 1.
3. *Additivity*: For mutually exclusive events $E_1, E_2, \ldots$

$$P\left[\bigcup_{i=1}^{\infty} E_i\right] = \sum_{i=1}^{\infty} P[E_i]$$

Using these axioms, many additional probability rules can easily be derived.

1.3 Probability Rules

Given an event A, and its complement, A^c, that is, the outcomes in Ω which are not in A, we have the **complement rule**:

$$P[A^c] = 1 - P[A]$$

In general, for two events A and B, we have the **addition rule**:

$$P[A \cup B] = P[A] + P[B] - P[A \cap B]$$

If A and B are also *disjoint*, then we have:

$$P[A \cup B] = P[A] + P[B]$$

If we have n mutually exclusive events, $E_1, E_2, \ldots E_n$, then we have:

$$P\left[\bigcup_{i=1}^{n} E_i\right] = \sum_{i=1}^{n} P[E_i]$$

Often, we would like to understand the probability of an event A, given some information about the outcome of event B. In that case, we have the **conditional probability rule** provided $P[B] > 0$.

$$P[A \mid B] = \frac{P[A \cap B]}{P[B]}$$

Rearranging the conditional probability rule, we obtain the **multiplication rule**:

$$P[A \cap B] = P[B] \cdot P[A \mid B]\cdot$$

For a number of events $E_1, E_2, \ldots E_n$, the multiplication rule can be expanded into the **chain rule**:

$$P\left[\bigcap_{i=1}^{n} E_i\right] = P[E_1] \cdot P[E_2 \mid E_1] \cdot P[E_3 \mid E_1 \cap E_2] \cdots P\left[E_n \mid \bigcap_{i=1}^{n-1} E_i\right]$$

Define a **partition** of a sample space Ω to be a set of disjoint events $A_1, A_2, \ldots, A_n$ whose union is the sample space Ω. That is

$$A_i \cap A_j = \emptyset$$

for all $i \neq j$, and

$$\bigcup_{i=1}^{n} A_i = \Omega.$$

Now, let $A_1, A_2, \ldots, A_n$ form a partition of the sample space where $P[A_i] > 0$ for all i. Then for any event B with $P[B] > 0$ we have **Bayes' Rule**:

$$P[A_i|B] = \frac{P[A_i]P[B|A_i]}{P[B]} = \frac{P[A_i]P[B|A_i]}{\sum_{i=1}^{n} P[A_i]P[B|A_i]}$$

The denominator of the latter equality is often called the **law of total probability**:

$$P[B] = \sum_{i=1}^{n} P[A_i]P[B|A_i]$$

Two events A and B are said to be **independent** if they satisfy

$$P[A \cap B] = P[A] \cdot P[B]$$

This becomes the new multiplication rule for independent events.

A collection of events $E_1, E_2, \ldots E_n$ is said to be independent if

$$P\left[\bigcup_{i \in S} E_i\right] = \prod_{i \in S} P[A_i]$$

for every subset S of $\{1, 2, \ldots n\}$.

If this is the case, then the chain rule is greatly simplified to:

$$P\left[\bigcap_{i=1}^{n} E_i\right] = \prod_{i=1}^{n} P[A_i]$$

1.4 Random Variables

A **random variable** is simply a *function* which maps outcomes in the sample space to real numbers.

1.4.1 Distributions

We often talk about the **distribution** of a random variable, which can be thought of as:

$$\text{distribution} = \text{list of possible } \textbf{values} + \text{associated } \textbf{probabilities}$$

This is not a strict mathematical definition, but is useful for conveying the idea.

If the possible values of a random variables are *discrete*, it is called a *discrete random variable*. If the possible values of a random variables are *continuous*, it is called a *continuous random variable*.

1.4.2 Discrete Random Variables

The distribution of a discrete random variable X is most often specified by a list of possible values and a probability **mass** function, $p(x)$. The mass function directly gives probabilities, that is,

$$p(x) = p_X(x) = P[X = x].$$

Note we almost always drop the subscript from the more correct $p_X(x)$ and simply refer to $p(x)$. The relevant random variable is discerned from context

The most common example of a discrete random variable is a **binomial** random variable. The mass function of a binomial random variable X, is given by

$$p(x|n,p) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x = 0, 1, \ldots, n, \ n \in \mathbb{N}, \ 0 < p < 1.$$

This line conveys a large ammount of information.

- The function $p(x|n,p)$ is the mass function. It is a function of x, the possible values of the random variable X. It is conditional on the **paramters** n and p. Different values of these parameters specify different binomial distributions.
- $x = 0, 1, \ldots, n$ indicates the **sample space**, that is, the possible values of the random variable.
- $n \in \mathbb{N}$ and $0 < p < 1$ specify the **parameter spaces**. These are the possible values of the parameters that give a valid binomial distribution.

Often all of this information is simply encoded by writing

$$X \sim \text{bin}(n,p).$$

1.4.3 Continuous Random Variables

The distribution of a continuous random variable X is most often specified by a set of possible values and a probability **density** function, $f(x)$. (A cumulative density or moment generating function would also suffice.)

The probability of the event $a < X < b$ is calculated as

$$P[a < X < b] = \int_a^b f(x)dx.$$

Note that densities are **not** probabilities.

The most common example of a continuous random variable is a **normal** random variable. The density of a normal random variable X, is given by

$$f(x|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left[\frac{-1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right], \quad -\infty < x < \infty, \ -\infty < \mu < \infty, \ \sigma > 0.$$

- The function $f(x|\mu,\sigma^2)$ is the density function. It is a function of x, the possible values of the random variable X. It is conditional on the **paramters** μ and σ^2. Different values of these parameters specify different normal distributions.
- $-\infty < x < \infty$ indicates the sample space. In this case, the random variable may take any value on the real line.
- $-\infty < \mu < \infty$ and $\sigma > 0$ specify the parameter space. These are the possible values of the parameters that give a valid normal distribution.

Often all of this information is simply encoded by writing

$$X \sim N(\mu, \sigma^2)$$

1.4.4 Several Random Variables

Consider two random variables X and Y. We say they are independent if

$$f(x, y) = f(x) \cdot f(y)$$

for all x and y. Here $f(x, y)$ is the **joint** density (mass) function of X and Y. We call $f(x)$ the **marginal** density (mass) function of X. Then $f(y)$ the marginal density (mass) function of Y. The joint density (mass) function $f(x, y)$ together with the possible (x, y) values specify the joint distribution of X and Y.

Similar notions exist for more than two variables.

1.5 Expectations

For discrete random variables, we define the **expectation** of the function of a random variable X as follows.

$$\mathbb{E}[g(X)] \triangleq \sum_x g(x)p(x)$$

For continuous random variables we have a similar definition.

$$\mathbb{E}[g(X)] \triangleq \int g(x)f(x)dx$$

For specific functions g, expectations are given names.

The **mean** of a random variable X is given by

$$\mu_X = \text{mean}[X] \triangleq \mathbb{E}[X].$$

So for a discrete random variable, we would have

$$\text{mean}[X] = \sum_x x \cdot p(x)$$

For a continuous random variable we would simply replace the sum by an integral.

The **variance** of a random variable X is given by

$$\sigma_X^2 = \text{var}[X] \triangleq \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2.$$

The **standard deviation of a random variable X is given by

$$\sigma_X = \text{sd}[X] \triangleq \sqrt{\sigma_X^2} = \sqrt{\text{var}[X]}.$$

The **covariance** or random variables X and Y is given by

$$\text{cov}[X, Y] \triangleq \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X] \cdot \mathbb{E}[Y].$$

1.6 Likelihood

Consider n iid random variables $X_1, X_2, \ldots X_n$. We can then write their **likelihood** as

$$\mathcal{L}(\theta \mid x_1, x_2, \ldots x_n) = \prod_{i=i}^{n} f(x_i; \theta)$$

where $f(x_i; \theta)$ is the density (or mass) function of random variable X_i evaluated at x_i with parameter θ.

Whereas a probability is a function of a possible observed value given a particular parameter value, a likelihood is the opposite. It is a function of a possible parameter value given observed data.

Maximumizing likelihood is a common techinque for fitting a model to data.

1.7 References

Any of the following are either dedicated to, or contain a good coverage of the details of the topics above.

- Probability Texts
  - Introduction to Probability by Dimitri P. Bertsekas and John N. Tsitsiklis
  - A First Course in Probability by Sheldon Ross
- Machine Learning Texts with Probability Focus
  - Probability for Statistics and Machine Learning by Anirban DasGupta
  - Machine Learning: A Probabilistic Perspective by Kevin P. Murphy
- Statistics Texts with Introduction to Probability
  - Probability and Statistical Inference by Robert V. Hogg, Elliot Tanis, and Dale Zimmerman
  - Introduction to Mathematical Statistics by Robert V. Hogg, Joseph McKean, and Allen T. Craig

Chapter 2

Introduction to R

> "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."
>
> — **Bill Gates**

After reading this chapter you will be able to:

- Interact with R using RStudio.
- Use R as a calculator.
- Work with data as vectors and data frames.
- Make basic data visualizations.
- Write your own R functions.
- Perform hypothesis tests using R.
- Perform basic simulations in R.

2.1 R Resources

R is both a programming language and software environment for statistical computing, which is *free* and *open-source*. To get started, you will need to install two pieces of software:

- R, the actual programming language.
  - Chose your operating system, and select the most recent version, 3.3.2.
- RStudio, an excellent IDE for working with R.
  - Note, you must have R installed to use RStudio. RStudio is simply an interface used to interact with R.

The popularity of R is on the rise, and everyday it becomes a better tool for statistical analysis. It even generated this book! (A skill you will learn in this course.) There are many good resources for learning R. They are not necessary for this course, but you may find them useful if you would like a deeper understanding of R:

- Try R from Code School.
  - An interactive introduction to the basics of R. Could be very useful for getting up to speed on R's syntax.

- Quick-R by Robert Kabacoff.
  - A good reference for R basics.
- R Tutorial by Chi Yau.
  - A combination reference and tutorial for R basics.
- R Markdown from RStudio.
  - Reference materials for RMarkdown.
- The Art of R Programming by Norman Matloff.
  - Gentle introduction to the programming side of R. (Whereas we will focus more on the data analysis side.) A free electronic version is available through the Illinois library.
- Advanced R by Hadley Wickham.
  - From the author of several extremely popular R packages. Good follow-up to The Art of R Programming. (And more up-to-date material.)
- R for Data Science by Hadley Wickham and Garrett Grolemund.
  - Similar to Advanced R, but focuses more on data analysis, while still introducing programming concepts. At the time of writing, currently under development.
- The R Inferno by Patrick Burns.
  - Likens learning the tricks of R to descending through the levels of hell. Very advanced material, but may be important if R becomes a part of your everyday toolkit.

RStudio has a large number of useful keyboard shortcuts. A list of these can be found using a keyboard shortcut – the keyboard shortcut to rule them all:

- On Windows: `Alt` + `Shift` + `K`
- On Mac: `Option` + `Shift` + `K`

The RStudio team has developed a number of "cheatsheets" for working with both R and RStudio. This particular cheatseet for Base R will summarize many of the concepts in this document.

When programming, it is often a good practice to follow a style guide. (Where do spaces go? Tabs or spaces? Underscores or CamelCase when naming variables?) No style guide is "correct" but it helps to be aware of what others do. The more import thing is to be consistent within your own code.

- Hadley Wickham Style Guide from Advanced R
- Google Style Guide

For this course, our main deviation from these two guides is the use of `=` in place of `<-`. (More on that later.)

2.2 R Basics

2.2.1 Basic Calculations

To get started, we'll use R like a simple calculator.

Addition, Subtraction, Multiplication and Division

Math	R	Result
$3+2$	`3 + 2`	5
$3-2$	`3 - 2`	1
$3 \cdot 2$	`3 * 2`	6
$3/2$	`3 / 2`	1.5

Exponents

Math	R	Result
3^2	`3 ^ 2`	9
$2^{(-3)}$	`2 ^ (-3)`	0.125
$100^{1/2}$	`100 ^ (1 / 2)`	10
$\sqrt{100}$	`sqrt(100)`	10

Mathematical Constants

Math	R	Result
π	`pi`	3.1415927
e	`exp(1)`	2.7182818

Logarithms

Note that we will use ln and log interchangeably to mean the natural logarithm. There is no `ln()` in R, instead it uses `log()` to mean the natural logarithm.

Math	R	Result
$\log(e)$	`log(exp(1))`	1
$\log_{10}(1000)$	`log10(1000)`	3
$\log_2(8)$	`log2(8)`	3
$\log_4(16)$	`log(16, base = 4)`	2

Trigonometry

Math	R	Result
$\sin(\pi/2)$	`sin(pi / 2)`	1
$\cos(0)$	`cos(0)`	1

2.2.2 Getting Help

In using R as a calculator, we have seen a number of functions: `sqrt()`, `exp()`, `log()` and `sin()`. To get documentation about a function in R, simply put a question mark in front of the function name and RStudio will display the documentation, for example:

```
?log
?sin
?paste
?lm
```

Frequently one of the most difficult things to do when learning R is asking for help. First, you need to decide to ask for help, then you need to know *how* to ask for help. Your very first line of defense should be to Google your error message or a short description of your issue. (The ability to solve problems using this method is quickly becoming an extremely valuable skill.) If that fails, and it eventually will, you should ask for help. There are a number of things you should include when emailing an instructor, or posting to a help website such as Stack Exchange.

- Describe what you expect the code to do.
- State the end goal you are trying to achieve. (Sometimes what you expect the code to do, is not what you want to actually do.)
- Provide the full text of any errors you have received.
- Provide enough code to recreate the error. Often for the purpose of this course, you could simply email your entire `.R` or `.Rmd` file.
- Sometimes it is also helpful to include a screenshot of your entire RStudio window when the error occurs.

If you follow these steps, you will get your issue resolved much quicker, and possibly learn more in the process. Do not be discouraged by running into errors and difficulties when learning R. (Or any technical skill.) It is simply part of the learning process.

2.2.3 Installing Packages

R comes with a number of built-in functions and datasets, but one of the main strengths of R as an open-source project is its package system. Packages add additional functions and data. Frequently if you want to do something in R, and it isn't available by default, there is a good chance that there is a package that will fulfill your needs.

To install a package, use the `install.packages()` function. Think of this as buying a recipe book from the store, bringing it home, and putting it on your shelf.

```
install.packages("ggplot2")
```

Once a package is installed, it must be loaded into your current R session before being used. Think of this as taking the book off of the shelf and opening it up to read.

```
library(ggplot2)
```

Once you close R, all the packages are closed and put back on the imaginary shelf. The next time you open R, you do not have to install the package again, but you do have to load any packages you intend to use by invoking `library()`.

2.2.4 Data Types

R has a number of basic data *types*.

- Numeric

  - Also known as Double. The default type when dealing with numbers.
  - Examples: `1`, `1.0`, `42.5`
- Integer
  - Examples: `1L`, `2L`, `42L`
- Complex
  - Example: `4 + 2i`
- Logical
  - Two possible values: `TRUE` and `FALSE`
  - You can also use `T` and `F`, but this is *not* recommended.
  - `NA` is also considered logical.
- Character
  - Examples: `"a"`, `"Statistics"`, `"1 plus 2."`

`R` also has a number of basic data *structures*. A data structure is either homogeneous (all elements are of the same data type) or heterogeneous (elements can be of more than one data type).

Dimension	**Homogeneous**	**Heterogeneous**
1	Vector	List
2	Matrix	Data Frame
3+	Array	

2.2.5 Vectors

Many operations in `R` make heavy use of **vectors**. Vectors in `R` are indexed starting at `1`. That is what the `[1]` in the output is indicating, that the first element of the row being displayed is the first element of the vector. Larger vectors will start additional rows with `[*]` where `*` is the index of the first element of the row.

Possibly the most common way to create a vector in `R` is using the `c()` function, which is short for "combine."" As the name suggests, it combines a list of numbers separated by commas.

```
c(1, 3, 5, 7, 8, 9)
```

```
## [1] 1 3 5 7 8 9
```

Here `R` simply outputs this vector. If we would like to store this vector in a **variable** we can do so with the **assignment** operator `=`. In this case the variable `x` now holds the vector we just created, and we can access the vector by typing `x`.

```
x = c(1, 3, 5, 7, 8, 9)
x
```

```
## [1] 1 3 5 7 8 9
```

As an aside, there is a long history of the assignment operator in `R`, partially due to the keys available on the keyboards of the creators of the `S` language. (Which preceded `R`.) For simplicity we will use `=`, but know that often you will see `<-` as the assignment operator.

The pros and cons of these two are well beyond the scope of this book, but know that for our purposes you will have no issue if you simply use `=`. If you are interested in the weird cases where the difference matters, check out The R Inferno.

If you wish to use `<-`, you will still need to use `=`, however only for argument passing. Some users like to keep assignment (`<-`) and argument passing (`=`) separate. No matter what you choose, the more important thing is that you **stay consistent**. Also, if working on a larger collaborative project, you should use whatever style is already in place.

Frequently you may wish to create a vector based on a sequence of numbers. The quickest and easiest way to do this is with the `:` operator, which creates a sequence of integers between two specified integers.

```
(y = 1:100)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Here we see `R` labeling the rows after the first since this is a large vector. Also, we see that by putting parentheses around the assignment, `R` both stores the vector in a variable called `y` and automatically outputs `y` to the console.

To subset a vector, we use square brackets, `[]`.

```
x
```

```
## [1] 1 3 5 7 8 9
```

```
x[1]
```

```
## [1] 1
```

```
x[3]
```

```
## [1] 5
```

We see that `x[1]` returns the first element, and `x[3]` returns the third element.

```
x[-2]
```

```
## [1] 1 5 7 8 9
```

We can also exclude certain indexes, in this case the second element.

```
x[1:3]
```

```
## [1] 1 3 5
```

```
x[c(1,3,4)]
```

```
## [1] 1 5 7
```

Lastly we see that we can subset based on a vector of indices.

One of the biggest strengths of R is its use of vectorized operations. (Frequently the lack of understanding of this concept leads of a belief that R is *slow*. R is not the fastest language, but it has a reputation for being slower than it really is.)

```
x = 1:10
x + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
2 * x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```
2 ^ x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
log(x)
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

We see that when a function like `log()` is called on a vector `x`, a vector is returned which has applied the function to each element of the vector `x`.

```
vec_1 = 1:10
vec_2 = 1:1000
vec_3 = 42
```

The length of a vector can be obtained with the `length()` function.

```
length(vec_1)
```

```
## [1] 10
```

```
length(vec_2)
```

```
## [1] 1000
```

```
length(vec_3)
```

```
## [1] 1
```

Note that scalars do not exists in R. They are simply vectors of length 1.

If we want to create a sequence that isn't limited to integers and increasing by 1 at a time, we can use the seq() function.

```
seq(from = 1.5, to = 4.2, by = 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
## [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

We will discuss functions in detail later, but note here that the input labels from, to, and by are optional.

```
seq(1.5, 4.2, 0.1)
```

```
##  [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1
## [18] 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
```

Another common operation to create a vector is rep(), which can repeat a single value a number of times.

```
rep("A", times = 10)
```

```
##  [1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"
```

Or, rep() can be used to repeat a vector a number of times.

```
rep(x, times = 3)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  1  2  3
## [24]  4  5  6  7  8  9 10
```

We have now seen four different ways to create vectors:

- c()
- :
- seq()
- rep()

So far we have mostly used them in isolation, but they are often used together.

```
c(x, rep(seq(1, 9, 2), 3), c(1, 2, 3), 42, 2:4)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10  1  3  5  7  9  1  3  5  7  9  1  3  5
## [24]  7  9  1  2  3 42  2  3  4
```

2.2.6 Summary Statistics

R has built in functions for a large number of summary statistics.

```
y
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Central Tendency

Measure	R	Result
Mean	`mean(y)`	50.5
Median	`median(y)`	50.5

Spread

Measure	R	Result
Variance	`var(y)`	841.6666667
Standard Deviation	`sd(y)`	29.011492
IQR	`IQR(y)`	49.5
Minimum	`min(y)`	1
Maximum	`max(y)`	100
Range	`range(y)`	1, 100

2.2.7 Matrices

R can also be used for **matrix** calculations. Matrices have rows and columns containing a single data type. In a matrix, the order of rows and columns is important. (This is not true of *data frames*, which we will see later.)

Matrices can be created using the `matrix` function.

```
x = 1:9
x
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
X = matrix(x, nrow = 3, ncol = 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Note here that we are using two different variables: lower case `x`, which stores a vector and capital `X`, which stores a matrix. (Following the usual mathematical convention.) We can do this because `R` is case sensitive.

By default the `matrix` function reorders a vector into columns, but we can also tell `R` to use rows instead.

```
Y = matrix(x, nrow = 3, ncol = 3, byrow = TRUE)
Y
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

We can also create a matrix of a specified dimension where every element is the same, in this case `0`.

```
Z = matrix(0, 2, 4)
Z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
```

Like vectors, matrices can be subsetted using square brackets, `[]`. However, since matrices are two-dimensional, we need to specify both a row and a column when subsetting.

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
X[1, 2]
```

```
## [1] 4
```

Here we accessed the element in the first row and the second column. We could also subset an entire row or column.

```
X[1, ]
```

```
## [1] 1 4 7
```

```
X[, 2]
```

```
## [1] 4 5 6
```

We can also use vectors to subset more than one row or column at a time. Here we subset to the first and third column of the second row.

```
X[2, c(1, 3)]
```

```
## [1] 2 8
```

Matrices can also be created by combining vectors as columns, using `cbind`, or combining vectors as rows, using `rbind`.

```
x = 1:9
rev(x)
```

```
## [1] 9 8 7 6 5 4 3 2 1
```

```
rep(1, 9)
```

```
## [1] 1 1 1 1 1 1 1 1 1
```

```
cbind(x, rev(x), rep(1, 9))
```

```
##       x
##  [1,] 1 9 1
##  [2,] 2 8 1
##  [3,] 3 7 1
##  [4,] 4 6 1
##  [5,] 5 5 1
##  [6,] 6 4 1
##  [7,] 7 3 1
##  [8,] 8 2 1
##  [9,] 9 1 1
```

```
rbind(x, rev(x), rep(1, 9))
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## x    1    2    3    4    5    6    7    8    9
##      9    8    7    6    5    4    3    2    1
##      1    1    1    1    1    1    1    1    1
```

R can then be used to perform matrix calculations.

```
x = 1:9
y = 9:1
X = matrix(x, 3, 3)
Y = matrix(y, 3, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
Y
```

```
##      [,1] [,2] [,3]
## [1,]    9    6    3
## [2,]    8    5    2
## [3,]    7    4    1
```

```
X + Y
```

```
##      [,1] [,2] [,3]
## [1,]   10   10   10
## [2,]   10   10   10
## [3,]   10   10   10
```

```
X - Y
```

```
##      [,1] [,2] [,3]
## [1,]   -8   -2    4
## [2,]   -6    0    6
## [3,]   -4    2    8
```

```
X * Y
```

```
##      [,1] [,2] [,3]
## [1,]    9   24   21
## [2,]   16   25   16
## [3,]   21   24    9
```

```
X / Y
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.1111111 0.6666667 2.333333
## [2,] 0.2500000 1.0000000 4.000000
## [3,] 0.4285714 1.5000000 9.000000
```

Note that `X * Y` is not matrix multiplication. It is element by element multiplication. (Same for `X / Y`). Instead, matrix multiplication uses `%*%`. Other matrix functions include `t()` which gives the transpose of a matrix and `solve()` which returns the inverse of a square matrix if it is invertible.

```
X %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   90   54   18
## [2,]  114   69   24
## [3,]  138   84   30
```

```
t(X)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
Z = matrix(c(9, 2, -3, 2, 4, -2, -3, -2, 16), 3, byrow = TRUE)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    9    2   -3
## [2,]    2    4   -2
## [3,]   -3   -2   16
```

```
solve(Z)
```

```
##             [,1]        [,2]       [,3]
## [1,]  0.12931034 -0.05603448 0.01724138
## [2,] -0.05603448  0.29094828 0.02586207
## [3,]  0.01724138  0.02586207 0.06896552
```

R has a number of matrix specific functions for obtaining dimension and summary information.

```
X = matrix(1:6, 2, 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(X)
```

```
## [1] 2 3
```

```
rowSums(X)
```

```
## [1]  9 12
```

```
colSums(X)
```

```
## [1]  3  7 11
```

```
rowMeans(X)
```

```
## [1] 3 4
```

```
colMeans(X)
```

```
## [1] 1.5 3.5 5.5
```

The `diag()` function can be used in a number of ways. We can extract the diagonal of a matrix.

```
diag(Z)
```

```
## [1]  9  4 16
```

Or create a matrix with specified elements on the diagonal. (And `0` on the off-diagonals.)

```
diag(1:5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

Or, lastly, create a square matrix of a certain dimension with `1` for every element of the diagonal and `0` for the off-diagonals.

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

Calculations with Vectors and Matrices

Certain operations in `R`, for example `%*%` have different behavior on vectors and matrices. To illustrate this, we will first create two vectors.

```
a_vec = c(1, 2, 3)
b_vec = c(2, 2, 2)
```

Note that these are indeed vectors. They are not matrices.

```
c(is.vector(a_vec), is.vector(b_vec))
```

```
## [1] TRUE TRUE
```

```
c(is.matrix(a_vec), is.matrix(b_vec))
```

```
## [1] FALSE FALSE
```

When this is the case, the `%*%` operator is used to calculate the **dot product**, also know as the **inner product** of the two vectors.

The dot product of vectors $\boldsymbol{a} = [a_1, a_2, \cdots a_n]$ and $\boldsymbol{b} = [b_1, b_2, \cdots b_n]$ is defined to be

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots a_n b_n.$$

```
a_vec %*% b_vec # inner product
```

```
##      [,1]
## [1,]   12
```

```
a_vec %o% b_vec # outer product
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

The `%o%` operator is used to calculate the **outer product** of the two vectors.

When vectors are coerced to become matrices, they are column vectors. So a vector of length n becomes an $n \times 1$ matrix after coercion.

```
as.matrix(a_vec)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

If we use the `%*%` operator on matrices, `%*%` again performs the expected matrix multiplication. So you might expect the following to produce an error, because the dimensions are incorrect.

```
as.matrix(a_vec) %*% b_vec
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

At face value this is a 3×1 matrix, multiplied by a 3×1 matrix. However, when `b_vec` is automatically coerced to be a matrix, `R` decided to make it a "row vector", a 1×3 matrix, so that the multiplication has conformable dimensions.

If we had coerced both, then `R` would produce an error.

```
as.matrix(a_vec) %*% as.matrix(b_vec)
```

Another way to calculate a *dot product* is with the `crossprod()` function. Given two vectors, the `crossprod()` function calculates their dot product. The function has a rather misleading name.

```
crossprod(a_vec, b_vec)  # inner product
```

```
##      [,1]
## [1,]   12
```

```
tcrossprod(a_vec, b_vec)  # outer product
```

```
##      [,1] [,2] [,3]
## [1,]    2    2    2
## [2,]    4    4    4
## [3,]    6    6    6
```

These functions could be very useful later. When used with matrices X and Y as arguments, it calculates

$$X^\top Y.$$

When dealing with linear models, the calculation

$$X^\top X$$

is used repeatedly.

```
C_mat = matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
D_mat = matrix(c(2, 2, 2, 2, 2, 2), 2, 3)
```

This is useful both as a shortcut for a frequent calculation and as a more efficient implementation than using `t()` and `%*%`.

```
crossprod(C_mat, D_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```
t(C_mat) %*% D_mat
```

```
##      [,1] [,2] [,3]
## [1,]    6    6    6
## [2,]   14   14   14
## [3,]   22   22   22
```

```
all.equal(crossprod(C_mat, D_mat), t(C_mat) %*% D_mat)
```

```
## [1] TRUE
```

```
crossprod(C_mat, C_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```
t(C_mat) %*% C_mat
```

```
##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```
all.equal(crossprod(C_mat, C_mat), t(C_mat) %*% C_mat)
```

```
## [1] TRUE
```

2.2.8 Data Frames

We have previously seen vectors and matrices for storing data as we introduced R. We will now introduce a **data frame** which will be the most common way that we store and interact with data in this course.

```
example_data = data.frame(x = c(1, 3, 5, 7, 9, 1, 3, 5, 7, 9),
                          y = rep("Hello", 10),
                          z = rep(c("TRUE", "FALSE"), 5))
```

Unlike a matrix, which can be thought of as a vector rearranged into rows and columns, a data frame is not required to have the same data type for each element. A data frame is a **list** of vectors. So, each vector must contain the same data type, but the different vectors can store different data types.

```
example_data
```

```
##    x     y     z
## 1  1 Hello  TRUE
## 2  3 Hello FALSE
## 3  5 Hello  TRUE
## 4  7 Hello FALSE
## 5  9 Hello  TRUE
## 6  1 Hello FALSE
## 7  3 Hello  TRUE
## 8  5 Hello FALSE
## 9  7 Hello  TRUE
## 10 9 Hello FALSE
```

The `data.frame()` function above is one way to create a data frame. We can also import data from various file types in into `R`, as well as use data stored in packages.

The example data above can also be found here as a .csv file. To read this data into `R`, we would use the `read_csv()` function from the `readr` package. Note that `R` has a built in function `read.csv()` that operates very similarly. The `readr` function `read_csv()` has a number of advantages. For example, it is much faster reading larger data. It also uses the `tibble` package to read the data as a tibble.

```
library(readr)
example_data_from_csv = read_csv("data/example_data.csv")
```

This particular line of code assumes that the file `example_data.csv` exists in a folder called `data` in your current working directory.

```
example_data_from_csv
```

```
## # A tibble: 10 × 3
##        x     y     z
##    <int> <chr> <lgl>
## 1      1 Hello  TRUE
## 2      3 Hello FALSE
## 3      5 Hello  TRUE
## 4      7 Hello FALSE
## 5      9 Hello  TRUE
## 6      1 Hello FALSE
## 7      3 Hello  TRUE
## 8      5 Hello FALSE
## 9      7 Hello  TRUE
## 10     9 Hello FALSE
```

A tibble is simply a data frame that prints with sanity. Notice in the output above that we are given additional information such as dimension and variable type.

The `as_tibble()` function can be used to coerce a regular data frame to a tibble.

```
library(tibble)
example_data = as_tibble(example_data)
example_data
```

```
## # A tibble: 10 × 3
##        x      y      z
##    <dbl> <fctr> <fctr>
## 1      1  Hello   TRUE
## 2      3  Hello  FALSE
## 3      5  Hello   TRUE
## 4      7  Hello  FALSE
## 5      9  Hello   TRUE
## 6      1  Hello  FALSE
## 7      3  Hello   TRUE
## 8      5  Hello  FALSE
## 9      7  Hello   TRUE
## 10     9  Hello  FALSE
```

Alternatively, we could use the "Import Dataset" feature in RStudio which can be found in the environment window. (By default, the top-right pane of RStudio.) Once completed, this process will automatically generate the code to import a file. The resulting code will be shown in the console window. In recent versions of RStudio, `read_csv()` is used by default, thus reading in a tibble.

Earlier we looked at installing packages, in particular the `ggplot2` package. (A package for visualization. While not necessary for this course, it is quickly growing in popularity.)

```
library(ggplot2)
```

Inside the `ggplot2` package is a dataset called `mpg`. By loading the package using the `library()` function, we can now access `mpg`.

When using data from inside a package, there are three things we would generally like to do:

- Look at the raw data.
- Understand the data. (Where did it come from? What are the variables? Etc.)
- Visualize the data.

To look at the data, we have two useful commands: `head()` and `str()`.

```
head(mpg, n = 10)
```

```
## # A tibble: 10 × 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   1.8  1999     4   auto(l5)     f    18    29
## 2          audi         a4   1.8  1999     4 manual(m5)     f    21    29
## 3          audi         a4   2.0  2008     4 manual(m6)     f    20    31
## 4          audi         a4   2.0  2008     4   auto(av)     f    21    30
## 5          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 6          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 7          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 8          audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
## 9          audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
## 10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
## # ... with 2 more variables: fl <chr>, class <chr>
```

The function `head()` will display the first `n` observations of the data frame. The `head()` function was more useful before tibbles. Notice that `mpg` is a tibble already, so the output from `head()` indicates there are only `10` observations. Note that this applies to `head(mpg, n = 10)` and not `mpg` itself. Also note that tibbles print a limited number of rows and columns by default. The last line of the printed output indicates with rows and columns were omitted.

```
mpg
```

```
## # A tibble: 234 × 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
## 1          audi         a4   1.8  1999     4   auto(l5)     f    18    29
## 2          audi         a4   1.8  1999     4 manual(m5)     f    21    29
## 3          audi         a4   2.0  2008     4 manual(m6)     f    20    31
## 4          audi         a4   2.0  2008     4   auto(av)     f    21    30
```

```
## 5          audi         a4   2.8  1999     6   auto(l5)     f    16    26
## 6          audi         a4   2.8  1999     6 manual(m5)     f    18    26
## 7          audi         a4   3.1  2008     6   auto(av)     f    18    27
## 8          audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
## 9          audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
## 10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

The function `str()` will display the "structure" of the data frame. It will display the number of **observations** and **variables**, list the variables, give the type of each variable, and show some elements of each variable. This information can also be found in the "Environment" window in RStudio.

```
str(mpg)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ class       : chr  "compact" "compact" "compact" "compact" ...
```

It is important to note that while matrices have rows and columns, data frames (tibbles) instead have observations and variables. When displayed in the console or viewer, each row is an observation and each column is a variable. However generally speaking, their order does not matter, it is simply a side-effect of how the data was entered or stored.

In this dataset an observation is for a particular model-year of a car, and the variables describe attributes of the car, for example its highway fuel efficiency.

To understand more about the data set, we use the `?` operator to pull up the documentation for the data.

```
?mpg
```

`R` has a number of functions for quickly working with and extracting basic information from data frames. To quickly obtain a vector of the variable names, we use the `names()` function.

```
names(mpg)
```

```
##  [1] "manufacturer" "model"        "displ"        "year"
##  [5] "cyl"          "trans"        "drv"          "cty"
##  [9] "hwy"          "fl"           "class"
```

To access one of the variables as a vector, we use the `$` operator.

```
mpg$year
```

```
##   [1] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
##  [15] 2008 1999 2008 2008 2008 2008 2008 1999 2008 1999 1999 2008 2008 2008
##  [29] 2008 2008 1999 1999 1999 2008 1999 2008 2008 1999 1999 1999 1999 2008
##  [43] 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008 2008 1999
##  [57] 1999 1999 2008 2008 2008 1999 2008 1999 2008 2008 2008 2008 2008 2008
##  [71] 1999 1999 2008 1999 1999 1999 2008 1999 1999 1999 2008 2008 1999 1999
##  [85] 1999 1999 1999 2008 1999 2008 1999 1999 2008 2008 1999 1999 2008 2008
##  [99] 2008 1999 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 2008
## [113] 1999 1999 2008 1999 1999 2008 2008 2008 2008 2008 2008 2008 1999 1999
## [127] 2008 2008 2008 2008 1999 2008 2008 1999 1999 1999 2008 1999 2008 2008
## [141] 1999 1999 1999 2008 2008 2008 2008 1999 1999 2008 1999 1999 2008 2008
## [155] 1999 1999 1999 2008 2008 1999 1999 2008 2008 2008 2008 1999 1999 1999
## [169] 1999 2008 2008 2008 2008 1999 1999 1999 1999 2008 2008 1999 1999 2008
## [183] 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008 1999 1999 1999
## [197] 2008 2008 1999 2008 1999 1999 2008 1999 1999 2008 2008 1999 1999 2008
## [211] 2008 1999 1999 1999 1999 2008 2008 2008 2008 1999 1999 1999 1999 1999
## [225] 1999 2008 2008 1999 1999 2008 2008 1999 1999 2008
```

```
mpg$hwy
```

```
##   [1] 29 29 31 30 26 26 27 26 25 28 27 25 25 25 25 24 25 23 20 15 20 17 17
##  [24] 26 23 26 25 24 19 14 15 17 27 30 26 29 26 24 24 22 22 24 24 17 22 21
##  [47] 23 23 19 18 17 17 19 19 12 17 15 17 17 12 17 16 18 15 16 12 17 17 16
##  [70] 12 15 16 17 15 17 17 18 17 19 17 19 19 17 17 17 16 16 17 15 17 26 25
##  [93] 26 24 21 22 23 22 20 33 32 32 29 32 34 36 36 29 26 27 30 31 26 26 28
## [116] 26 29 28 27 24 24 24 22 19 20 17 12 19 18 14 15 18 18 15 17 16 18 17
## [139] 19 19 17 29 27 31 32 27 26 26 25 25 17 17 20 18 26 26 27 28 25 25 24
## [162] 27 25 26 23 26 26 26 26 25 27 25 27 20 20 19 17 20 17 29 27 31 31 26
## [185] 26 28 27 29 31 31 26 26 27 30 33 35 37 35 15 18 20 20 22 17 19 18 20
## [208] 29 26 29 29 24 44 29 26 29 29 29 29 23 24 44 41 29 26 28 29 29 29 28
## [231] 29 26 26 26
```

We can use the `dim()`, `nrow()` and `ncol()` functions to obtain information about the dimension of the data frame.

```
dim(mpg)
```

```
## [1] 234  11
```

```
nrow(mpg)
```

```
## [1] 234
```

```
ncol(mpg)
```

```
## [1] 11
```

Here `nrow()` is also the number of observations, which in most cases is the *sample size*.

Subsetting data frames can work much like subsetting matrices using square brackets, `[,]`. Here, we find fuel efficient vehicles earning over 35 miles per gallon and only display `manufacturer`, `model` and `year`.

```
mpg[mpg$hwy > 35, c("manufacturer", "model", "year")]
```

```
## # A tibble: 6 × 3
##   manufacturer      model  year
##          <chr>      <chr> <int>
## 1        honda      civic  2008
## 2        honda      civic  2008
## 3       toyota    corolla  2008
## 4   volkswagen      jetta  1999
## 5   volkswagen new beetle  1999
## 6   volkswagen new beetle  1999
```

An alternative would be to use the `subset()` function, which has a much more readable syntax.

```
subset(mpg, subset = hwy > 35, select = c("manufacturer", "model", "year"))
```

Lastly, we could use the `filter` and `select` functions from the `dplyr` package which introduces the `%>%` operator from the `magrittr` package. This is not necessary for this course, however the `dplyr` package is something you should be aware of as it is becoming a popular tool in the `R` world.

```
library(dplyr)
mpg %>% filter(hwy > 35) %>% select(manufacturer, model, year)
```

All three approaches produce the same results. Which you use will be largely based on a given situation as well as user preference.

2.2.9 Plotting

Now that we have some data to work with, and we have learned about the data at the most basic level, our next tasks is to visualize the data. Often, a proper visualization can illuminate features of the data that can inform further analysis.

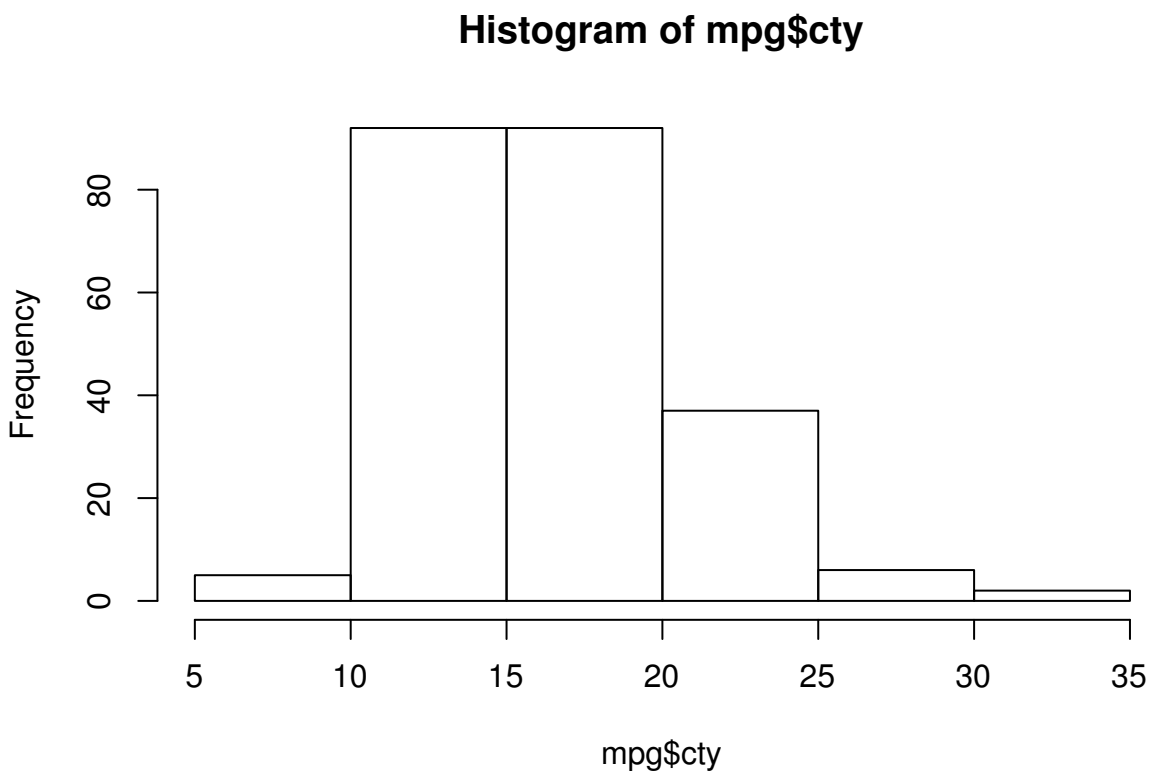
We will look at three methods of visualizing data that we will use throughout the course:

- Histograms
- Boxplots
- Scatterplots

2.2.9.1 Histograms

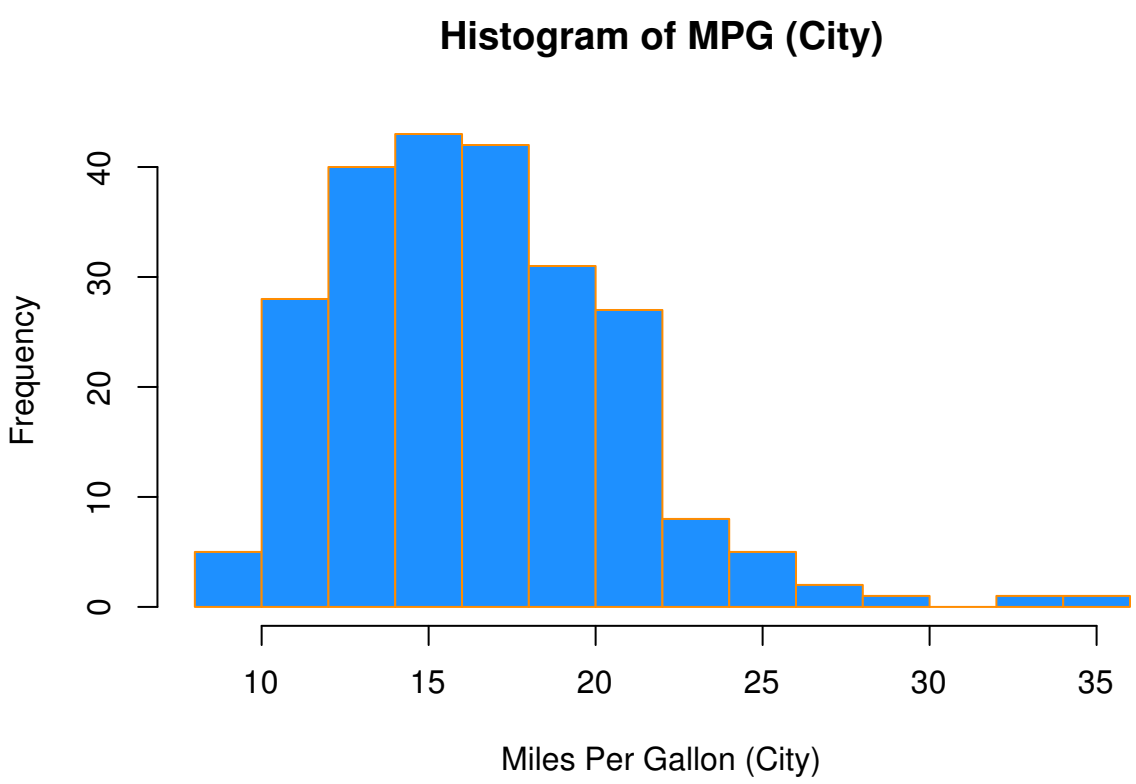
When visualizing a single numerical variable, a **histogram** will be our go-to tool, which can be created in `R` using the `hist()` function.

```
hist(mpg$cty)
```



The histogram function has a number of parameters which can be changed to make our plot look much nicer. Use the `?` operator to read the documentation for the `hist()` to see a full list of these parameters.

```
hist(mpg$cty,
     xlab   = "Miles Per Gallon (City)",
     main   = "Histogram of MPG (City)",
     breaks = 12,
     col    = "dodgerblue",
     border = "darkorange")
```



Importantly, you should always be sure to label your axes and give the plot a title. The argument `breaks` is specific to `hist()`. Entering an integer will give a suggestion to `R` for how many bars to use for the histogram. By default `R` will attempt to intelligently guess a good number of `breaks`, but as we can see here, it is sometimes useful to modify this yourself.

2.2.9.2 Boxplots

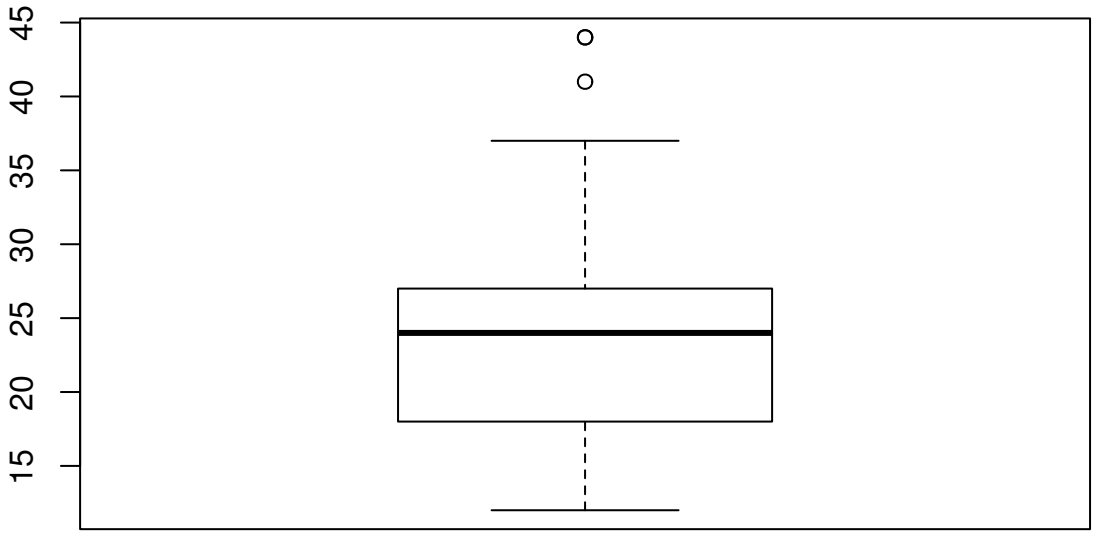
To visualize the relationship between a numerical and categorical variable, we will use a **boxplot**. In the `mpg` dataset, the `drv` variable takes a small, finite number of values. A car can only be front wheel drive, 4 wheel drive, or rear wheel drive.

```
unique(mpg$drv)
```

```
## [1] "f" "4" "r"
```

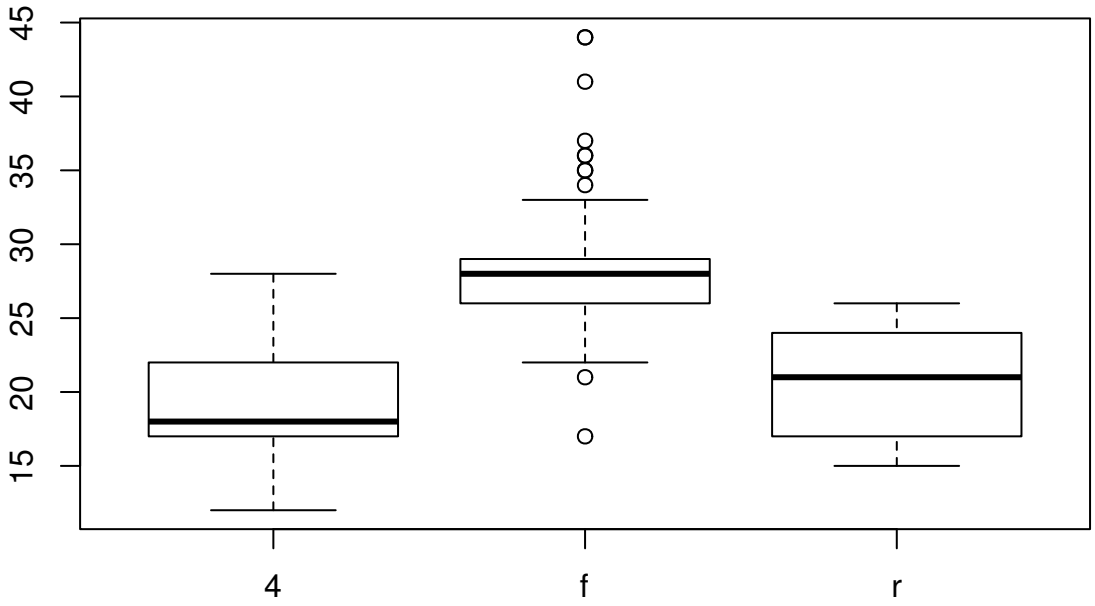
First note that we can use a single boxplot as an alternative to a histogram for visualizing a single numerical variable. To do so in `R`, we use the `boxplot()` function.

```
boxplot(mpg$hwy)
```



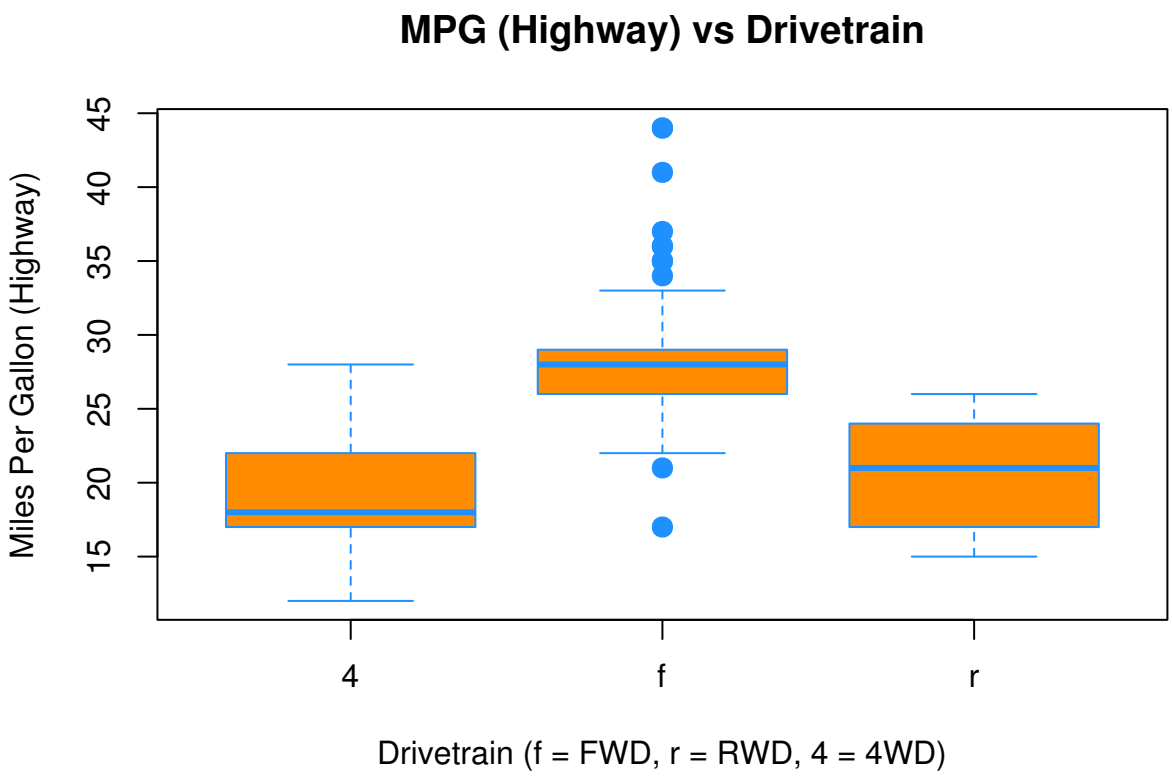
However, more often we will use boxplots to compare a numerical variable for different values of a categorical variable.

```
boxplot(hwy ~ drv, data = mpg)
```



Here used the `boxplot()` command to create side-by-side boxplots. However, since we are now dealing with two variables, the syntax has changed. The `R` syntax `hwy ~ drv, data = mpg` reads "Plot the `hwy` variable against the `drv` variable using the dataset `mpg`." We see the use of a `~` (which specifies a formula) and also a `data =` argument. This will be a syntax that is common to many functions we will use in this course.

```
boxplot(hwy ~ drv, data = mpg,
     xlab   = "Drivetrain (f = FWD, r = RWD, 4 = 4WD)",
     ylab   = "Miles Per Gallon (Highway)",
     main   = "MPG (Highway) vs Drivetrain",
     pch    = 20,
     cex    = 2,
     col    = "darkorange",
     border = "dodgerblue")
```

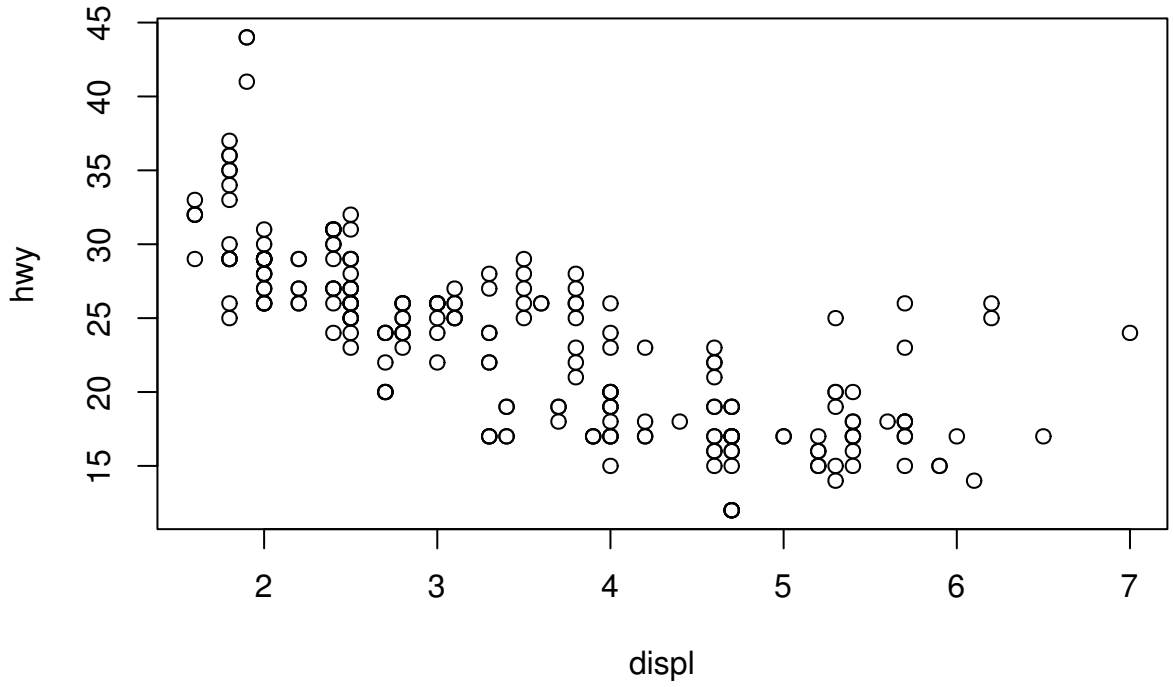


Again, `boxplot()` has a number of additional arguments which have the ability to make our plot more visually appealing.

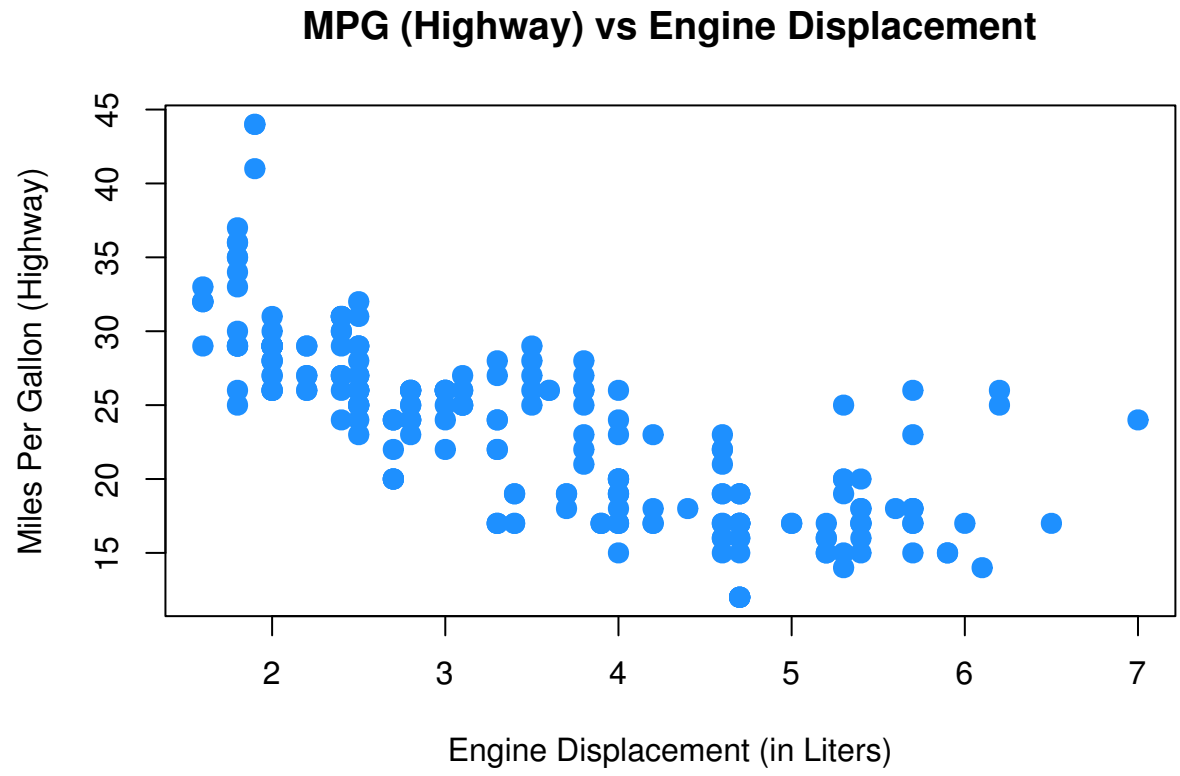
2.2.9.3 Scatterplots

Lastly, to visualize the relationship between two numeric variables we will use a **scatterplot**. This can be done with the `plot()` function and the `~` syntax we just used with a boxplot. (The function `plot()` can also be used more generally; see the documentation for details.)

```
plot(hwy ~ displ, data = mpg)
```




```
plot(hwy ~ displ, data = mpg,
     xlab = "Engine Displacement (in Liters)",
     ylab = "Miles Per Gallon (Highway)",
     main = "MPG (Highway) vs Engine Displacement",
     pch  = 20,
     cex  = 2,
     col  = "dodgerblue")
```



2.2.10 Distributions

When working with different statistical distributions, we often want to make probabilistic statements based on the distribution.

We typically want to know one of four things:

- The density (pdf) at a particular value.
- The distribution (cdf) at a particular value.
- The quantile value corresponding to a particular probability.
- A random draw of values from a particular distribution.

This used to be done with statistical tables printed in the back of textbooks. Now, R has functions for obtaining density, distribution, quantile and random values.

The general naming structure of the relevant R functions is:

- `dname` calculates density (pdf) at input `x`.
- `pname` calculates distribution (cdf) at input `x`.
- `qname` calculates the quantile at an input probability.
- `rname` generates a random draw from a particular distribution.

Note that `name` represents the name of the given distribution.

For example, consider a random variable X which is $N(\mu = 2, \sigma^2 = 25)$. (Note, we are parameterizing using the variance σ^2. R however uses the standard deviation.)

To calculate the value of the pdf at `x = 3`, that is, the height of the curve at `x = 3`, use:

```
dnorm(x = 3, mean = 2, sd = 5)
```

```
## [1] 0.07820854
```

To calculate the value of the cdf at `x = 3`, that is, $P(X \leq 3)$, the probability that X is less than or equal to 3, use:

```
pnorm(q = 3, mean = 2, sd = 5)
```

```
## [1] 0.5792597
```

Or, to calculate the quantile for probability 0.975, use:

```
qnorm(p = 0.975, mean = 2, sd = 5)
```

```
## [1] 11.79982
```

Lastly, to generate a random sample of size `n = 10`, use:

```
rnorm(n = 10, mean = 2, sd = 5)
```

```
##  [1]  1.194798 -1.603384 -2.895213 10.731478  5.956902  9.942161 10.574412
##  [8] -3.862256 -2.412222  8.568709
```

These functions exist for many other distributions, including but not limited to:

Command	Distribution
`*binom`	Binomial
`*t`	t
`*pois`	Poisson
`*f`	F
`*chisq`	Chi-Squared

Where `*` can be `d`, `p`, `q`, and `r`. Each distribution will have its own set of parameters which need to be passed to the functions as arguments. For example, `dbinom()` would not have arguments for `mean` and `sd`, since those are not parameters of the distribution. Instead a binomial distribution is usually parameterized by n and p, however `R` chooses to call them something else. To find the names that `R` uses we would use `?dbinom` and see that `R` instead calls the arguments `size` and `prob`. For example:

```
dbinom(x = 6, size = 10, prob = 0.75)
```

```
## [1] 0.145998
```

Also note that, when using the `dname` functions with discrete distributions, they are the pmf of the distribution. For example, the above command is $P(Y = 6)$ if $Y \sim b(n = 10, p = 0.75)$. (The probability of flipping an unfair coin `10` times and seeing `6` heads, if the probability of heads is `0.75`.)

2.3 Programming Basics

2.3.1 Logical Operators

Operator	Summary	Example	Result
`x < y`	x less than y	`3 < 42`	TRUE
`x > y`	x greater than y	`3 > 42`	FALSE
`x <= y`	x less than or equal to y	`3 <= 42`	TRUE
`x >= y`	x greater than or equal to y	`3 >= 42`	FALSE
`x == y`	xequal to y	`3 == 42`	FALSE
`x != y`	x not equal to y	`3 != 42`	TRUE
`!x`	not x	`!(3 > 42)`	TRUE
`x | y`	x or y	`(3 > 42) | TRUE`	TRUE
`x & y`	x and y	`(3 < 4) & ( 42 > 13)`	TRUE

In `R`, logical operators are vectorized. To demonstrate this, we will use the following height and weight data.

```
heights = c(110, 120, 115, 136, 205, 156, 175)
weights = c(64, 67, 62, 60, 77, 70, 66)
```

First, using the `<` operator, when can find which `heights` are less than `121`. Further, we could also find which `heights` are less than `121` or exactly equal to `156`.

```
heights < 121
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

```
heights < 121 | heights == 156
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

Often, a vector of logical values is useful for subsetting a vector. For example, we can find the `heights` that are larger than `150`. We can then use the resulting vector to subset the `heights` vector, thus actually returning the `heights` that are above `150`, instead of a vector of which values are above `150`. Here we also obtain the `weights` corresponding to `heights` above `150`.

```
heights > 150
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
heights[heights > 150]
```

```
## [1] 205 156 175
```

```
weights[heights > 150]
```

```
## [1] 77 70 66
```

When comparing vectors, be sure you are comparing vectors of the same length.

```
a = 1:10
b = 2:4
a < b
```

```
## Warning in a < b: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

What happened here? `R` still performed the operation, but it also gives us a warning. (To perform the operation, `R` automatically made `b` longer by repeating `b` as needed.)

The one exception to this behavior is comparing to a vector of length `1`. `R` does not warn us in this case, as comparing each value of a vector to a single value is a common operation that is usually reasonable to perform.

```
a > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Often we will want to convert `TRUE` and `FALSE` values to `1` and `0`. When performing mathematical operations on `TRUE` and `FALSE`, this is done automatically through type coercion.

```
5 + (a > 5)
```

```
##  [1] 5 5 5 5 5 6 6 6 6 6
```

By calling `sum()` on a vector of logical values, we can essentially count the number of `TRUE` values.

```
sum(a > 5)
```

```
## [1] 5
```

Here we count the elements of `a` that are larger than `5.` This is an extremely useful feature.

2.3.2 Control Flow

In `R`, the if/else syntax is:

```
if (...) {
  some R code
} else {
  more R code
}
```

For example,

```
x = 1
y = 3
if (x > y) {
  z = x * y
  print("x is larger than y")
} else {
  z = x + 5 * y
  print("x is less than or equal to y")
}
```

```
## [1] "x is less than or equal to y"
```

```
z
```

```
## [1] 16
```

`R` also has a special function `ifelse()` which is very useful. It returns one of two specified values based on a conditional statement.

```
ifelse(4 > 3, 1, 0)
```

```
## [1] 1
```

The real power of `ifelse()` comes from its ability to be applied to vectors.

```
fib = c(1, 1, 2, 3, 5, 8, 13, 21)
ifelse(fib > 6, "Foo", "Bar")
```

```
## [1] "Bar" "Bar" "Bar" "Bar" "Bar" "Foo" "Foo" "Foo"
```

Now a `for` loop example,

```
x = 11:15
for (i in 1:5) {
  x[i] = x[i] * 2
}

x
```

```
## [1] 22 24 26 28 30
```

Note that this `for` loop is very normal in many programming languages, but not in R. In R we would not use a loop, instead we would simply use a vectorized operation.

```
x = 11:15
x = x * 2
x
```

```
## [1] 22 24 26 28 30
```

2.3.3 Functions

So far we have been using functions, but haven't actually discussed some of their details.

```
function_name(arg1 = 10, arg2 = 20)
```

To use a function, you simply type its name, followed by an open parenthesis, then specify values of its arguments, then finish with a closing parenthesis.

An **argument** is a variable which is used in the body of the function. Specifying the values of the arguments is essentially providing the inputs to the function.

We can also write our own functions in R. For example, we often like to "standardize" variables, that is, subtracting the sample mean, and dividing by the sample standard deviation.

$$\frac{x - \bar{x}}{s}$$

In R we would write a function to do this. When writing a function, there are three thing you must do.

- Give the function a name. Preferably something that is short, but descriptive.
- Specify the arguments using `function()`
- Write the body of the function within curly braces, `{}`.

```
standardize = function(x) {
  m = mean(x)
  std = sd(x)
  result = (x - m) / std
  result
}
```

Here the name of the function is `standardize`, and the function has a single argument `x` which is used in the body of function. Note that the output of the final line of the body is what is returned by the function. In this case the function returns the vector stored in the variable `results`.

To test our function, we will take a random sample of size `n = 10` from a normal distribution with a mean of 2 and a standard deviation of 5.

```
(test_sample = rnorm(n = 10, mean = 2, sd = 5))
```

```
##  [1] -6.4335502 -1.0802619  2.2895098 -0.6148338 -2.0695388  3.8365832
##  [7]  0.6838855  7.4253202  3.2484777 -0.1386416
```

```
standardize(x = test_sample)
```

```
##  [1] -1.893640109 -0.475501660  0.417183848 -0.352205199 -0.737570794
##  [6]  0.827018737 -0.008161734  1.777710287  0.671223835 -0.226057211
```

This function could be written much more succinctly, simply performing all the operations on one line and immediately returning the result, without storing any of the intermediate results.

```
standardize = function(x) {
  (x - mean(x)) / sd(x)
}
```

When specifying arguments, you can provide default arguments.

```
power_of_num = function(num, power = 2) {
  num ^ power
}
```

Let's look at a number of ways that we could run this function to perform the operation `10^2` resulting in `100`.

```
power_of_num(10)
```

```
## [1] 100
```

```
power_of_num(10, 2)
```

```
## [1] 100
```

```
power_of_num(num = 10, power = 2)
```

```
## [1] 100
```

```
power_of_num(power = 2, num = 10)
```

```
## [1] 100
```

Note that without using the argument names, the order matters. The following code will not evaluate to the same output as the previous example.

```
power_of_num(2, 10)
```

```
## [1] 1024
```

Also, the following line of code would produce an error since arguments without a default value must be specified.

```
power_of_num(power = 5)
```

To further illustrate a function with a default argument, we will write a function that calculates sample variance two ways.

By default, is will calculate the unbiased estimate of σ^2, which we will call s^2.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x - \bar{x})^2$$

It will also have the ability to return the biased estimate (based on maximum likelihood) which we will call $\hat{\sigma}^2$.

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x - \bar{x})^2$$

```
get_var = function(x, biased = FALSE) {
  n = length(x) - 1 * !biased
  (1 / n) * sum((x - mean(x)) ^ 2)
}
```

```
get_var(test_sample)
```

```
## [1] 14.24964
```

```
get_var(test_sample, biased = FALSE)
```

```
## [1] 14.24964
```

```
var(test_sample)
```

```
## [1] 14.24964
```

We see the function is working as expected, and when returning the unbiased estimate it matches R's built in function `var()`. Finally, let's examine the biased estimate of σ^2.

```
get_var(test_sample, biased = TRUE)
```

```
## [1] 12.82468
```

2.4 Hypothesis Tests in R

A prerequisite for STAT 420 is an understanding of the basics of hypothesis testing. Recall the basic structure of hypothesis tests:

- An overall model and related assumptions are made. (The most common being observations following a normal distribution.)

- The **null** (H_0) and **alternative** (H_1 or H_A) hypothesis are specified. Usually the null specifies a particular value of a parameter.
- With given data, the **value** of the *test statistic* is calculated.
- Under the general assumptions, as well as assuming the null hypothesis is true, the **distribution** of the *test statistic* is known.
- Given the distribution and value of the test statistic, as well as the form of the alternative hypothesis, we can calculate a **p-value** of the test.
- Based on the **p-value** and pre-specified level of significance, we make a decision. One of:
  - Fail to reject the null hypothesis.
  - Reject the null hypothesis.

We'll do some quick review of two of the most common tests to show how they are performed using R.

2.4.1 One Sample t-Test: Review

Suppose $x_i \sim \text{N}(\mu, \sigma^2)$ and we want to test $H_0 : \mu = \mu_0$ versus $H_1 : \mu \neq \mu_0$.

Assuming σ is unknown, we use the one-sample Student's t test statistic:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \sim t_{n-1},$$

where $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ and $s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$.

A $100(1-\alpha)\%$ confidence interval for μ is given by,

$$\bar{x} \pm t_{n-1}(\alpha/2)\frac{s}{\sqrt{n}}$$

where $t_{n-1}(\alpha/2)$ is the critical value such that $P\left(t > t_{n-1}(\alpha/2)\right) = \alpha/2$ for $n-1$ degrees of freedom.

2.4.2 One Sample t-Test: Example

Suppose a grocery store sells "16 ounce" boxes of *Captain Crisp* cereal. A random sample of 9 boxes was taken and weighed. The weight in ounces are stored in the data frame `capt_crisp`.

```
capt_crisp = data.frame(weight = c(15.5, 16.2, 16.1, 15.8, 15.6, 16.0, 15.8, 15.9, 16.2))
```

The company that makes *Captain Crisp* cereal claims that the average weight of a box is at least 16 ounces. We will assume the weight of cereal in a box is normally distributed and use a 0.05 level of significance to test the company's claim.

To test $H_0 : \mu \geq 16$ versus $H_1 : \mu < 16$, the test statistic is

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

The sample mean $\bar{x}$ and the sample standard deviation s can be easily computed using R. We also create variables which store the hypothesized mean and the sample size.

```
x_bar = mean(capt_crisp$weight)
s     = sd(capt_crisp$weight)
mu_0  = 16
n     = 9
```

We can then easily compute the test statistic.

```
t = (x_bar - mu_0) / (s / sqrt(n))
t
```

```
## [1] -1.2
```

Under the null hypothesis, the test statistic has a t distribution with $n - 1$ degrees of freedom, in this case 8.

To complete the test, we need to obtain the p-value of the test. Since this is a one-sided test with a less-than alternative, we need to area to the left of -1.2 for a t distribution with 8 degrees of freedom. That is,

$$P(t_8 < -1.2)$$

```
pt(t, df = n - 1)
```

```
## [1] 0.1322336
```

We now have the p-value of our test, which is greater than our significance level (0.05), so we fail to reject the null hypothesis.

Alternatively, this entire process could have been completed using one line of R code.

```
t.test(x = capt_crisp$weight, mu = 16, alternative = c("less"), conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  capt_crisp$weight
## t = -1.2, df = 8, p-value = 0.1322
## alternative hypothesis: true mean is less than 16
## 95 percent confidence interval:
##      -Inf 16.05496
## sample estimates:
## mean of x
##      15.9
```

We supply R with the data, the hypothesized value of μ, the alternative, and the confidence level. R then returns a wealth of information including:

- The value of the test statistic.
- The degrees of freedom of the distribution under the null hypothesis.
- The p-value of the test.
- The confidence interval which corresponds to the test.
- An estimate of μ.

Since the test was one-sided, R returned a one-sided confidence interval. If instead we wanted a two-sided interval for the mean weight of boxes of *Captain Crisp* cereal we could modify our code.

```
capt_test_results = t.test(capt_crisp$weight, mu = 16,
                           alternative = c("two.sided"), conf.level = 0.95)
```

This time we have stored the results. By doing so, we can directly access portions of the output from `t.test()`. To see what information is available we use the `names()` function.

```
names(capt_test_results)
```

```
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

We are interested in the confidence interval which is stored in `conf.int`.

```
capt_test_results$conf.int
```

```
## [1] 15.70783 16.09217
## attr(,"conf.level")
## [1] 0.95
```

Let's check this interval "by hand." The one piece of information we are missing is the critical value, $t_{n-1}(\alpha/2) = t_8(0.025)$, which can be calculated in R using the `qt()` function.

```
qt(0.975, df = 8)
```

```
## [1] 2.306004
```

So, the 95% CI for the mean weight of a cereal box is calculated by plugging into the formula,

$$\bar{x} \pm t_{n-1}(\alpha/2)\frac{s}{\sqrt{n}}$$

```
c(mean(capt_crisp$weight) - qt(0.975, df = 8) * sd(capt_crisp$weight) / sqrt(9),
  mean(capt_crisp$weight) + qt(0.975, df = 8) * sd(capt_crisp$weight) / sqrt(9))
```

```
## [1] 15.70783 16.09217
```

2.4.3 Two Sample t-Test: Review

Suppose $x_i \sim \mathrm{N}(\mu_x, \sigma^2)$ and $y_i \sim \mathrm{N}(\mu_y, \sigma^2)$.

Want to test $H_0 : \mu_x - \mu_y = \mu_0$ versus $H_1 : \mu_x - \mu_y \neq \mu_0$.

Assuming σ is unknown, use the two-sample Student's t test statistic:

$$t = \frac{(\bar{x} - \bar{y}) - \mu_0}{s_p\sqrt{\frac{1}{n} + \frac{1}{m}}} \sim t_{n+m-2},$$

where $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$, $\bar{y} = \frac{\sum_{i=1}^{m} y_i}{m}$, and $s_p^2 = \frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}$.

A $100(1-\alpha)\%$ CI for $\mu_x - \mu_y$ is given by

$$(\bar{x} - \bar{y}) \pm t_{n+m-2}(\alpha/2) \left(s_p \sqrt{\tfrac{1}{n} + \tfrac{1}{m}} \right),$$

where $t_{n+m-2}(\alpha/2)$ is the critical value such that $P\left(t > t_{n+m-2}(\alpha/2)\right) = \alpha/2$.

2.4.4 Two Sample t-Test: Example

Assume that the distributions of X and Y are $\text{N}(\mu_1, \sigma^2)$ and $\text{N}(\mu_2, \sigma^2)$, respectively. Given the $n = 6$ observations of X,

```
x = c(70, 82, 78, 74, 94, 82)
n = length(x)
```

and the $m = 8$ observations of Y,

```
y = c(64, 72, 60, 76, 72, 80, 84, 68)
m = length(y)
```

we will test $H_0 : \mu_1 = \mu_2$ versus $H_1 : \mu_1 > \mu_2$.

First, note that we can calculate the sample means and standard deviations.

```
x_bar = mean(x)
s_x   = sd(x)
y_bar = mean(y)
s_y   = sd(y)
```

We can then calculate the pooled standard deviation.

$$s_p = \sqrt{\frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}}$$

```
s_p = sqrt(((n - 1) * s_x ^ 2 + (m - 1) * s_y ^ 2) / (n + m - 2))
```

Thus, the relevant t test statistic is given by

$$t = \frac{(\bar{x} - \bar{y}) - \mu_0}{s_p \sqrt{\frac{1}{n} + \frac{1}{m}}}.$$

```
t = ((x_bar - y_bar) - 0) / (s_p * sqrt(1 / n + 1 / m))
t
```

```
## [1] 1.823369
```

Note that $t \sim t_{n+m-2} = t_{12}$, so we can calculate the p-value, which is

$$P(t_{12} > 1.8233692).$$

```
1 - pt(t, df = n + m - 2)
```

```
## [1] 0.04661961
```

But, then again, we could have simply performed this test in one line of R.

```
t.test(x, y, alternative = c("greater"), var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  x and y
## t = 1.8234, df = 12, p-value = 0.04662
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.1802451       Inf
## sample estimates:
## mean of x mean of y
##        80        72
```

Recall that a two-sample t-test can be done with or without an equal variance assumption. Here `var.equal = TRUE` tells R we would like to perform the test under the equal variance assumption.

Above we carried out the analysis using two vectors `x` and `y`. In general, we will have a preference for using data frames.

```
t_test_data = data.frame(values = c(x, y),
                         group  = c(rep("A", length(x)), rep("B", length(y))))
```

We now have the data stored in a single variables (`values`) and have created a second variable (`group`) which indicates which "sample" the value belongs to.

```
t_test_data
```

```
##    values group
## 1      70     A
## 2      82     A
## 3      78     A
## 4      74     A
## 5      94     A
## 6      82     A
## 7      64     B
## 8      72     B
## 9      60     B
## 10     76     B
## 11     72     B
## 12     80     B
## 13     84     B
## 14     68     B
```

Now to perform the test, we still use the `t.test()` function but with the `~` syntax and a `data` argument.

```
t.test(values ~ group, data = t_test_data,
       alternative = c("greater"), var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  values by group
## t = 1.8234, df = 12, p-value = 0.04662
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.1802451       Inf
## sample estimates:
## mean in group A mean in group B
##              80              72
```

2.5 Simulation

Simulation and model fitting are related but opposite processes.

- In **simulation**, the *data generating process* is known. We will know the form of the model as well as the value of each of the parameters. In particular, we will often control the distribution and parameters which define the randomness, or noise in the data.
- In **model fitting**, the *data* is known. We will then assume a certain form of model and find the best possible values of the parameters given the observed data. Essentially we are seeking to uncover the truth. Often we will attempt to fit many models, and we will learn metrics to assess which model fits best.

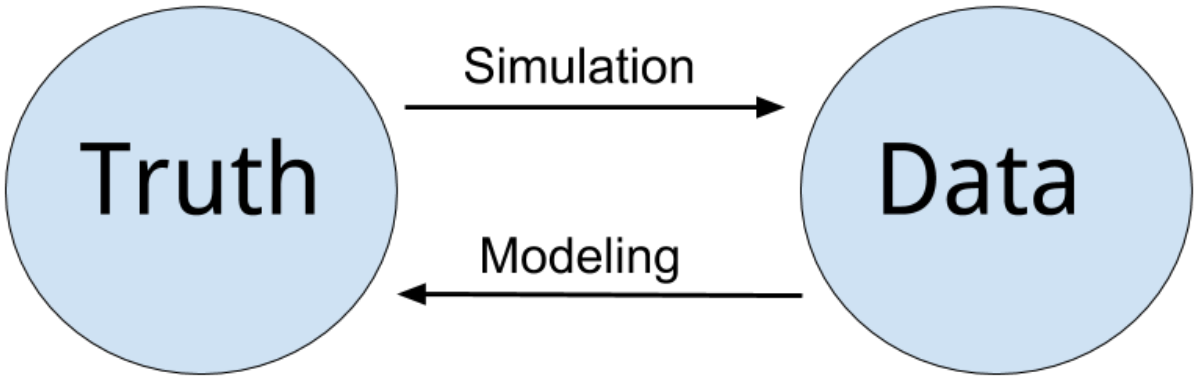


Figure 2.1: Simulation vs Modeling

Often we will simulate data according to a process we decide, then use a modeling method seen in class. We can then verify how well the method works, since we know the data generating process.

One of the biggest strengths of `R` is its ability to carry out simulations using built-in functions for generating random samples from certain distributions. We'll look at two very simple examples here, however simulation will be a topic we revisit several times throughout the course.

2.5.1 Paired Differences

Consider the model:

$$X_{11}, X_{12}, \ldots, X_{1n} \sim N(\mu_1, \sigma^2)$$
$$X_{21}, X_{22}, \ldots, X_{2n} \sim N(\mu_2, \sigma^2)$$

Assume that $\mu_1 = 6$, $\mu_2 = 5$, $\sigma^2 = 4$ and $n = 25$.

Let

$$\bar{X}_1 = \frac{1}{n}\sum_{i=1}^{n} X_{1i}$$
$$\bar{X}_2 = \frac{1}{n}\sum_{i=1}^{n} X_{2i}$$
$$D = \bar{X}_1 - \bar{X}_2.$$

Suppose we would like to calculate $P(0 < D < 2)$. First we will need to obtain the distribution of D.

Recall,

$$\bar{X}_1 \sim N\left(\mu_1, \frac{\sigma^2}{n}\right)$$

and

$$\bar{X}_2 \sim N\left(\mu_2, \frac{\sigma^2}{n}\right).$$

Then,

$$D = \bar{X}_1 - \bar{X}_2 \sim N\left(\mu_1 - \mu_2, \frac{\sigma^2}{n} + \frac{\sigma^2}{n}\right) = N\left(6 - 5, \frac{4}{25} + \frac{4}{25}\right).$$

So,

$$D \sim N(\mu = 1, \sigma^2 = 0.32).$$

Thus,

$$P(0 < D < 2) = P(D < 2) - P(D < 0).$$

This can then be calculated using `R` without a need to first standardize, or use a table.

```
pnorm(2, mean = 1, sd = sqrt(0.32)) - pnorm(0, mean = 1, sd = sqrt(0.32))
```

```
## [1] 0.9229001
```

An alternative approach, would be to **simulate** a large number of observations of D then use the **empirical distribution** to calculate the probability.

Our strategy will be to repeatedly:

- Generate a sample of 25 random observations from $N(\mu_1 = 6, \sigma^2 = 4)$. Call the mean of this sample $\bar{x}_{1s}$.
- Generate a sample of 25 random observations from $N(\mu_1 = 5, \sigma^2 = 4)$. Call the mean of this sample $\bar{x}_{2s}$.
- Calculate the differences of the means, $d_s = \bar{x}_{1s} - \bar{x}_{2s}$.

We will repeat the process a large number of times. Then we will use the distribution of the simulated observations of d_s as an estimate for the true distribution of D.

```
set.seed(42)
num_samples = 10000
differences = rep(0, num_samples)
```

Before starting our `for` loop to perform the operation, we set a seed for reproducibility, create and set a variable `num_samples` which will define the number of repetitions, and lastly create a variables `differences` which will store the simulate values, d_s.

By using `set.seed()` we can reproduce the random results of `rnorm()` each time starting from that line.

```
for (s in 1:num_samples) {
  x1 = rnorm(n = 25, mean = 6, sd = 2)
  x2 = rnorm(n = 25, mean = 5, sd = 2)
  differences[s] = mean(x1) - mean(x2)
}
```

To estimate $P(0 < D < 2)$ we will find the proportion of values of d_s (among the 10^{4} values of d_s generated) that are between 0 and 2.

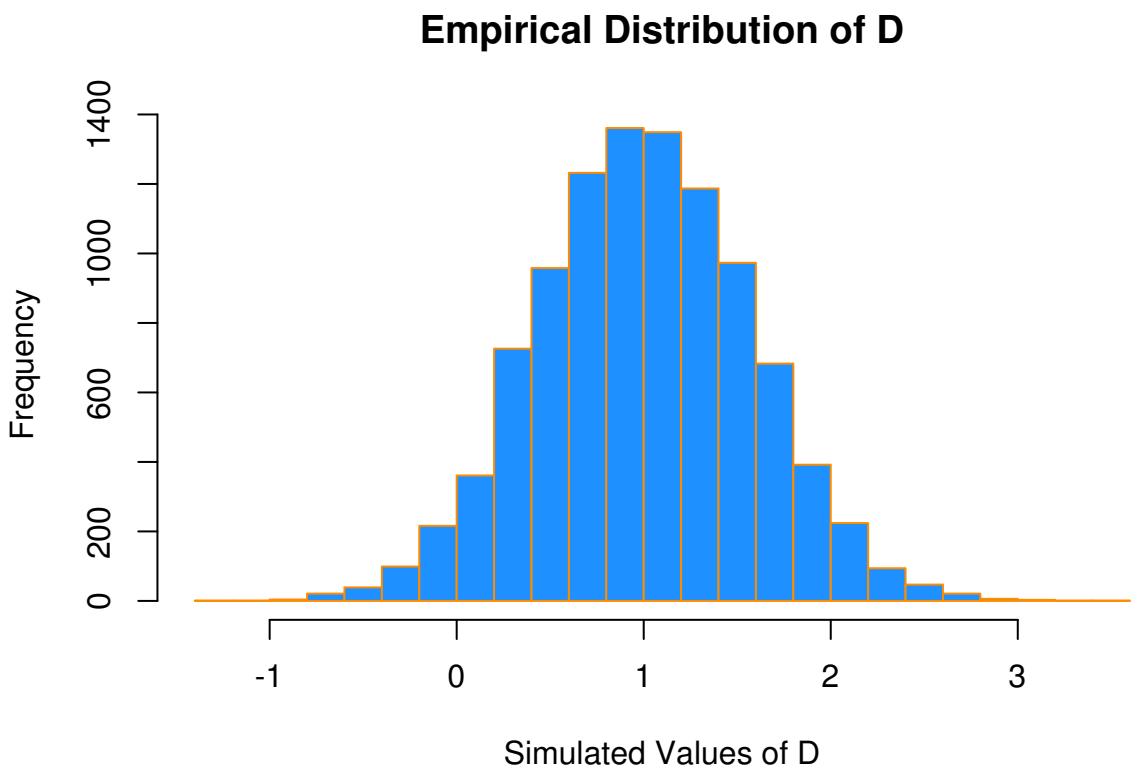
```
mean(0 < differences & differences < 2)
```

```
## [1] 0.9222
```

Recall that above we derived the distribution of D to be $N(\mu = 1, \sigma^2 = 0.32)$

If we look at a histogram of the differences, we find that it looks very much like a normal distribution.

```
hist(differences, breaks = 20,
     main   = "Empirical Distribution of D",
     xlab   = "Simulated Values of D",
     col    = "dodgerblue",
     border = "darkorange")
```



Also the sample mean and variance are very close to to what we would expect.

```
mean(differences)
```

```
## [1] 1.001423
```

```
var(differences)
```

```
## [1] 0.3230183
```

We could have also accomplished this task with a single line of more "idiomatic" `R`.

```
set.seed(42)
diffs = replicate(10000, mean(rnorm(25, 6, 2)) - mean(rnorm(25, 5, 2)))
```

Use `?replicate` to take a look at the documentation for the `replicate` function and see if you can understand how this line performs the same operations that our `for` loop above executed.

```
mean(differences == diffs)
```

```
## [1] 1
```

We see that by setting the same seed for the randomization, we actually obtain identical results!

2.5.2 Distribution of a Sample Mean

For another example of simulation, we will simulate observations from a Poisson distribution, and examine the empirical distribution of the sample mean of these observations.

Recall, if

$$X \sim Pois(\mu)$$

then

$$E[X] = \mu$$

and

$$Var[X] = \mu.$$

Also, recall that for a random variable X with finite mean μ and finite variance σ^2, the central limit theorem tells us that the mean, $\bar{X}$ of a random sample of size n is approximately normal for *large* values of n. Specifically, as $n \to \infty$,

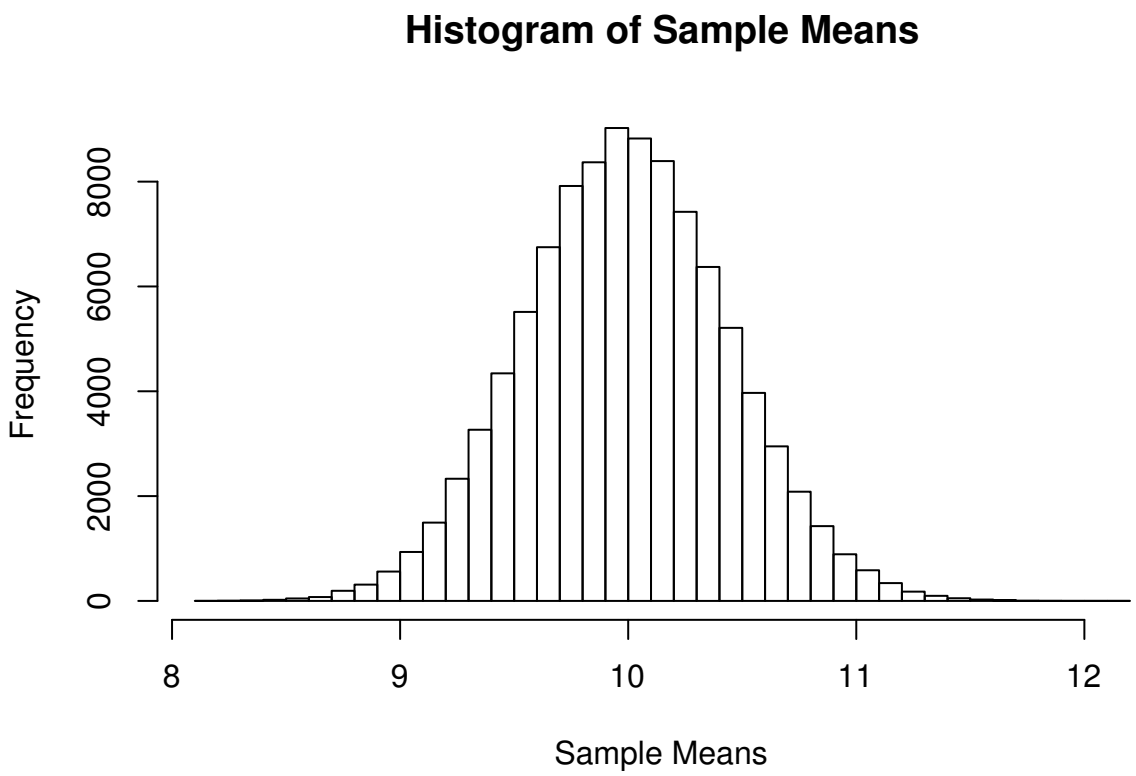
$$\bar{X} \xrightarrow{d} N\left(\mu, \frac{\sigma^2}{n}\right).$$

The following verifies this result for a Poisson distribution with $\mu = 10$ and a sample size of $n = 50$.

```
set.seed(1337)
mu          = 10
sample_size = 50
samples     = 100000
x_bars      = rep(0, samples)
```

```
for(i in 1:samples){
  x_bars[i] = mean(rpois(sample_size, lambda = mu))
}
```

```
x_bar_hist = hist(x_bars, breaks = 50,
                  main = "Histogram of Sample Means",
                  xlab = "Sample Means")
```



Now we will compare sample statistics from the empirical distribution with their known values based on the parent distribution.

```
c(mean(x_bars), mu)
```

```
## [1] 10.00008 10.00000
```

```
c(var(x_bars), mu / sample_size)
```

```
## [1] 0.1989732 0.2000000
```

```
c(sd(x_bars), sqrt(mu) / sqrt(sample_size))
```

```
## [1] 0.4460641 0.4472136
```

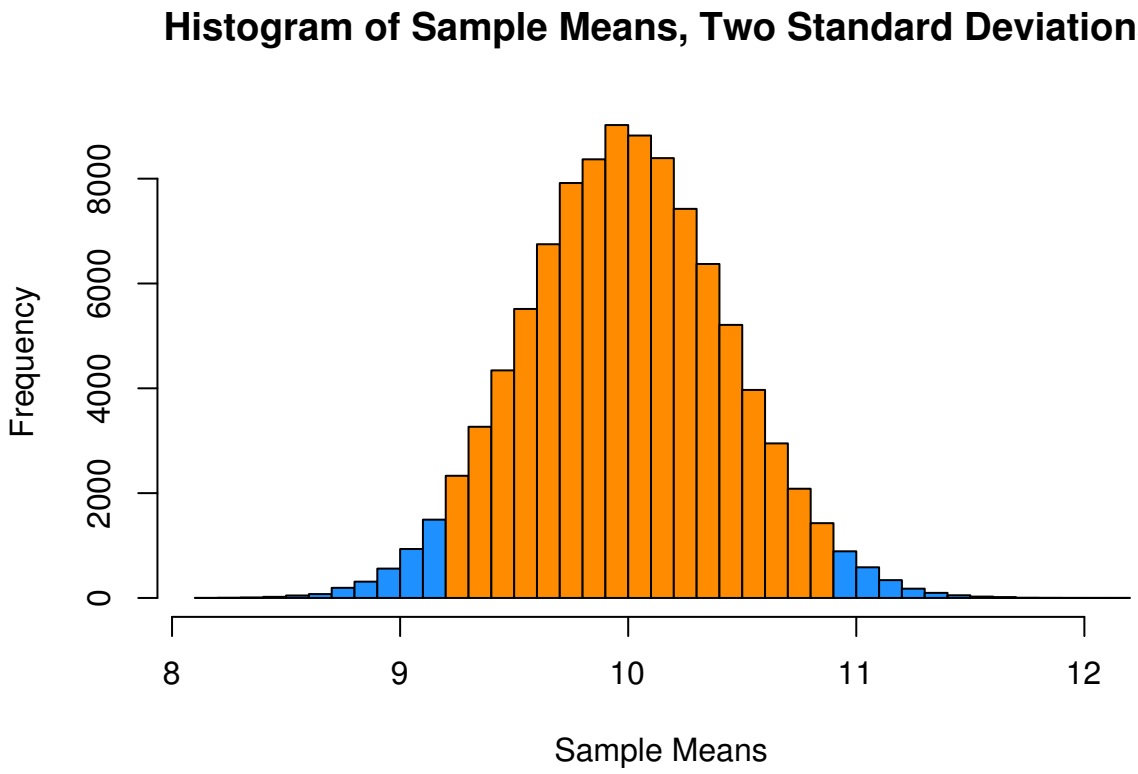
And here, we will calculate the proportion of sample means that are within 2 standard deviations of the population mean.

```
mean(x_bars > mu - 2 * sqrt(mu) / sqrt(sample_size) &
     x_bars < mu + 2 * sqrt(mu) / sqrt(sample_size))
```

```
## [1] 0.95429
```

This last histogram uses a bit of a trick to approximately shade the bars that are within two standard deviations of the mean.)

```
shading = ifelse(x_bar_hist$breaks > mu - 2 * sqrt(mu) / sqrt(sample_size) &
                   x_bar_hist$breaks < mu + 2 * sqrt(mu) / sqrt(sample_size),
                 "darkorange", "dodgerblue")

x_bar_hist = hist(x_bars, breaks = 50, col = shading,
                  main = "Histogram of Sample Means, Two Standard Deviations",
                  xlab = "Sample Means")
```



Chapter 3

RStudio and RMarkdown

This chapter will serve as a (currently brief) collection of tutorials for using RStudio and RMarkdown. It will likely be expanded over time. At this time many resources also appear in the previous chapter. Over time some may be moved here.

The following videos were made as an introduction to `R`, RStudio, and RMarkdown for STAT 420 at UIUC.

- RStudio Basics
- RMarkdown Intro
- RMarkdown Basics
- RMarkdown Tips and Tricks

Note that RStudio and RMarkdown and constantly receiving excellent support and updates, so these videos already contain some outdated information. For example, as of recent RStudio versions, the "Import Dataset" functionality has been updated to utilize the `readr` and `tibble` packages. Additionally, working interactively with RMarkdown documents in RStudio has a long list of new functionality.

RStudio provides their own tutorial for RMarkdown. They also have an excellent RStudio "cheatsheets" which visually identifies many of the features available in the IDE.

3.1 Template

This `.zip` file contains the files necessary to produce this rendered document. This document is a more complete version of a template than what is seen in the above videos.

Chapter 4

Regression Basics in R

This chapter will recap the basics of performing regression analyses in R. For more detailed coverage, see Applied Statistics with R.

We will use the Advertising data associated with Introduction to Statistical Learning.

```
library(readr)
Advertising = read_csv("data/Advertising.csv")
```

After loading data into R, our first step should **always** be to inspect the data. We will start by simply printing some observations in order to understand the basic structure of the data.

```
Advertising
```

```
## # A tibble: 200 × 4
##       TV Radio Newspaper Sales
##    <dbl> <dbl>     <dbl> <dbl>
## 1  230.1  37.8      69.2  22.1
## 2   44.5  39.3      45.1  10.4
## 3   17.2  45.9      69.3   9.3
## 4  151.5  41.3      58.5  18.5
## 5  180.8  10.8      58.4  12.9
## 6    8.7  48.9      75.0   7.2
## 7   57.5  32.8      23.5  11.8
## 8  120.2  19.6      11.6  13.2
## 9    8.6   2.1       1.0   4.8
## 10 199.8   2.6      21.2  10.6
## # ... with 190 more rows
```

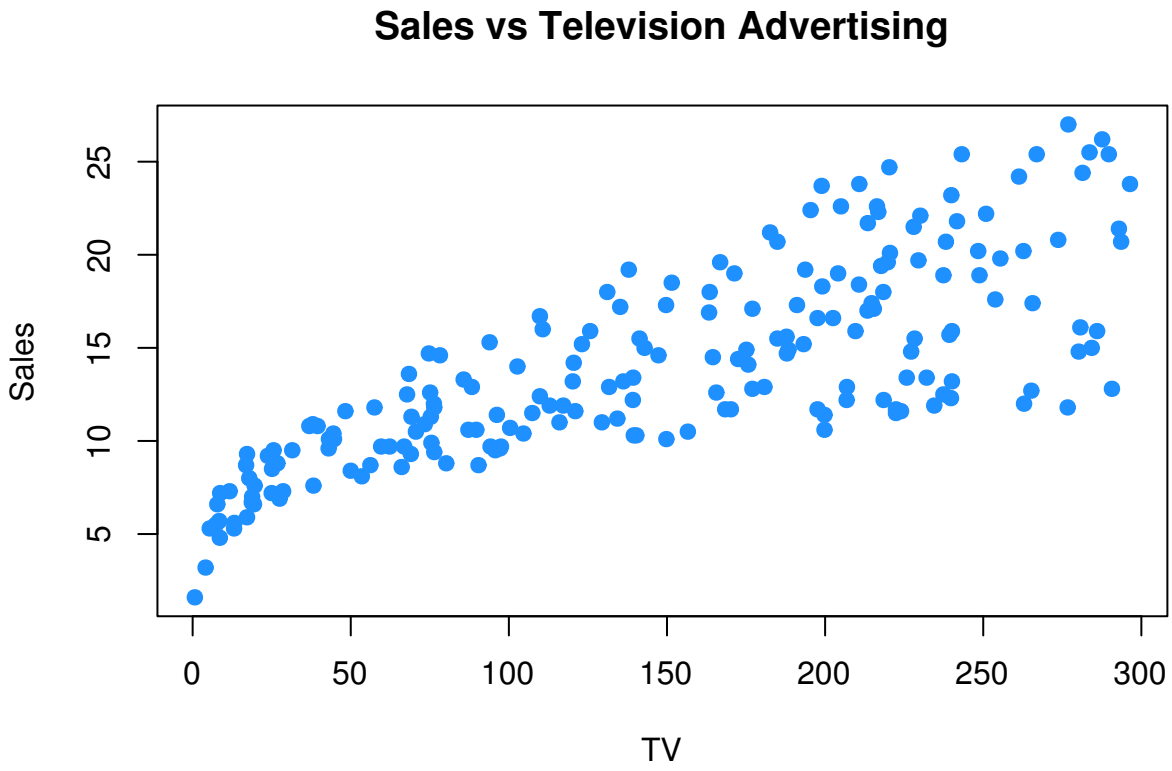
Because the data was read using `read_csv()`, `Advertising` is a tibble. We see that there are a total of 200 observations and 4 variables, each of which is numeric. (Specifically double-precision vectors, but more importantly they are numbers.) For the purpose of this analysis, `Sales` will be the **response variable**. That is, we seek to understand the relationship between `Sales`, and the **predictor variables**: `TV`, `Radio`, and `Newspaper`.

4.1 Visualization for Regression

After investigating the structure of the data, the next step should be to visualize the data. Since we have only numeric variables, we should consider **scatter plots**.

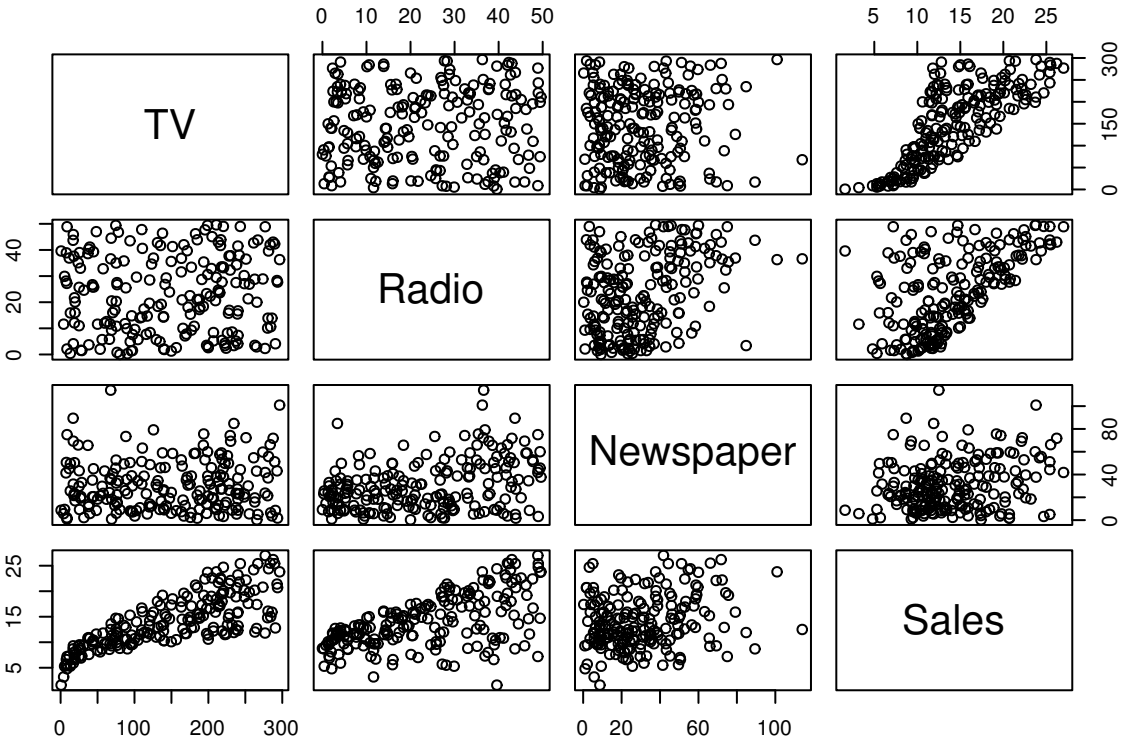
We could do so for any individual predictor.

```
plot(Sales ~ TV, data = Advertising, col = "dodgerblue", pch = 20, cex = 1.5,
     main = "Sales vs Television Advertising")
```



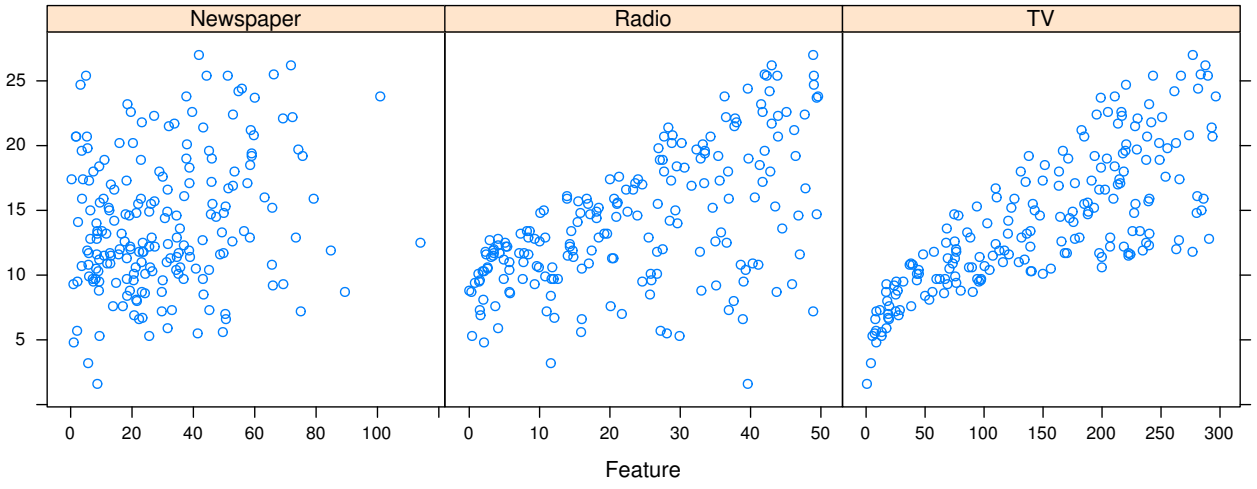
The `pairs()` function is a useful way to quickly visualize a number of scatter plots.

```
pairs(Advertising)
```



Often, we will be most interested in only the relationship between each predictor and the response. For this, we can use the `featurePlot()` function from the `caret` package. (We will use the `caret` package more and more frequently as we introduce new topics.)

```
library(caret)
featurePlot(x = Advertising[ , c("TV", "Radio", "Newspaper")], y = Advertising$Sales)
```



We see that there is a clear increase in `Sales` as `Radio` or `TV` are increased. The relationship between `Sales` and `Newspaper` is less clear. How all of the predictors work together is also unclear, as there is some obvious correlation between `Radio` and `TV`. To investigate further, we will need to model the data.

4.2 The lm() Function

The following code fits an additive **linear model** with `Sales` as the response and each remaining variable as a predictor. Note, by not using `attach()` and instead specifying the `data =` argument, we are able to specify this model without using each of the variable names directly.

```
mod_1 = lm(Sales ~ ., data = Advertising)
# mod_1 = lm(Sales ~ TV + Radio + Newspaper, data = Advertising)
```

Note that the commented line is equivalent to the line that is run, but we will often use the `response ~ .` syntax when possible.

4.3 Hypothesis Testing

The `summary()` function will return a large amount of useful information about a model fit using `lm()`. Much of it will be helpful for hypothesis testing including individual tests about each predictor, as well as the significance of the regression test.

```
summary(mod_1)
```

```
##
## Call:
## lm(formula = Sales ~ ., data = Advertising)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.8277 -0.8908  0.2418  1.1893  2.8292
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.938889   0.311908   9.422   <2e-16 ***
## TV           0.045765   0.001395  32.809   <2e-16 ***
## Radio        0.188530   0.008611  21.893   <2e-16 ***
## Newspaper   -0.001037   0.005871  -0.177     0.86
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.686 on 196 degrees of freedom
## Multiple R-squared:  0.8972, Adjusted R-squared:  0.8956
## F-statistic: 570.3 on 3 and 196 DF,  p-value: < 2.2e-16
```

```
mod_0 = lm(Sales ~ TV + Radio, data = Advertising)
```

The `anova()` function is useful for comparing two models. Here we compare the full additive model, `mod_1`, to a reduced model `mod_0`. Essentially we are testing for the significance of the `Newspaper` variable in the additive model.

```
anova(mod_0, mod_1)
```

```
## Analysis of Variance Table
##
## Model 1: Sales ~ TV + Radio
## Model 2: Sales ~ TV + Radio + Newspaper
##   Res.Df    RSS Df Sum of Sq      F Pr(>F)
## 1    197 556.91
## 2    196 556.83  1  0.088717 0.0312 0.8599
```

Note that hypothesis testing is *not* our focus, so we omit many details.

4.4 Prediction

The `predict()` function is an extremely versatile function, for, prediction. When used on the result of a model fit using `lm()` it will, by default, return predictions for each of the data points used to fit the model. (Here, we limit the printed result to the first 10.)

```
head(predict(mod_1), n = 10)
```

```
##         1         2         3         4         5         6         7
## 20.523974 12.337855 12.307671 17.597830 13.188672 12.478348 11.729760
##         8         9        10
## 12.122953  3.727341 12.550849
```

Note that the effect of the `predict()` function is dependent on the input to the function. Here, we are supplying as the first argument a model object of class `lm`. Because of this, `predict()` then runs the `predict.lm()` function. Thus, we should use `?predict.lm()` for details.

We could also specify new data, which should be a data frame or tibble with the same column names as the predictors.

```
new_obs = data.frame(TV = 150, Radio = 40, Newspaper = 1)
```

We can then use the `predict()` function for point estimates, confidence intervals, and prediction intervals.

Using only the first two arguments, `R` will simply return a point estimate, that is, the "predicted value," $\hat{y}$.

```
predict(mod_1, newdata = new_obs)
```

```
##        1
## 17.34375
```

If we specify an additional argument `interval` with a value of `"confidence"`, `R` will return a 95% confidence interval for the mean response at the specified point. Note that here `R` also gives the point estimate as `fit`.

```
predict(mod_1, newdata = new_obs, interval = "confidence")
```

```
##        fit      lwr      upr
## 1 17.34375 16.77654 17.91096
```

Lastly, we can alter the level using the `level` argument. Here we report a prediction interval instead of a confidence interval.

```
predict(mod_1, newdata = new_obs, interval = "prediction", level = 0.99)
```

```
##        fit      lwr      upr
## 1 17.34375 12.89612 21.79138
```

4.5 Unusual Observations

R provides several functions for obtaining metrics related to unusual observations.

- `resid()` provides the residual for each observation
- `hatvalues()` gives the leverage of each observation
- `rstudent()` give the studentized residual for each observation
- `cooks.distance()` calculates the influence of each observation

```
head(resid(mod_1), n = 10)
```

```
##           1           2           3           4           5           6
##  1.57602559 -1.93785482 -3.00767078  0.90217049 -0.28867186 -5.27834763
##           7           8           9          10
##  0.07024005  1.07704683  1.07265914 -1.95084872
```

```
head(hatvalues(mod_1), n = 10)
```

```
##           1           2           3           4           5           6
## 0.025202848 0.019418228 0.039226158 0.016609666 0.023508833 0.047481074
##           7           8           9          10
## 0.014435091 0.009184456 0.030714427 0.017147645
```

```
head(rstudent(mod_1), n = 10)
```

```
##           1           2           3           4           5           6
##  0.94680369 -1.16207937 -1.83138947  0.53877383 -0.17288663 -3.28803309
##           7           8           9          10
##  0.04186991  0.64099269  0.64544184 -1.16856434
```

```
head(cooks.distance(mod_1), n = 10)
```

```
##            1            2            3            4            5
## 5.797287e-03 6.673622e-03 3.382760e-02 1.230165e-03 1.807925e-04
##            6            7            8            9           10
## 1.283058e-01 6.452021e-06 9.550237e-04 3.310088e-03 5.945006e-03
```

4.6 Adding Complexity

We have a number of ways to add complexity to a linear model, even allowing a linear model to be used to model non-linear relationships.

4.6.1 Interactions

Interactions can be introduced to the `lm()` procedure in a number of ways.

We can use the `:` operator to introduce a single interaction of interest.

```
mod_2 = lm(Sales ~ . + TV:Newspaper, data = Advertising)
coef(mod_2)
```

```
##   (Intercept)            TV         Radio     Newspaper  TV:Newspaper
##  3.8730824491  0.0392939602  0.1901312252 -0.0320449675  0.0002016962
```

The `response ~ . ^ k` syntax can be used to model all `k`-way interactions. (As well as the appropriate lower order terms.) Here we fit a model with all two-way interactions, and the lower order main effects.

```
mod_3 = lm(Sales ~ . ^ 2, data = Advertising)
coef(mod_3)
```

```
##     (Intercept)              TV           Radio       Newspaper
##    6.460158e+00    2.032710e-02    2.292919e-02    1.703394e-02
##        TV:Radio    TV:Newspaper Radio:Newspaper
##    1.139280e-03   -7.971435e-05   -1.095976e-04
```

The `*` operator can be used to specify all interactions of a certain order, as well as all lower order terms according to the usual hierarchy. Here we see a three-way interaction and all lower order terms.

```
mod_4 = lm(Sales ~ TV * Radio * Newspaper, data = Advertising)
coef(mod_4)
```

```
##        (Intercept)                 TV              Radio
##       6.555887e+00       1.971030e-02       1.962160e-02
##          Newspaper           TV:Radio       TV:Newspaper
##       1.310565e-02       1.161523e-03      -5.545501e-05
##    Radio:Newspaper TV:Radio:Newspaper
##       9.062944e-06      -7.609955e-07
```

Note that, we have only been dealing with numeric predictors. **Categorical predictors** are often recorded as **factor** variables in `R`.

```
library(tibble)
cat_pred = tibble(
  x1 = factor(c(rep("A", 10), rep("B", 10), rep("C", 10))),
  x2 = runif(n = 30),
  y  = rnorm(n = 30)
)
cat_pred
```

```
## # A tibble: 30 × 3
##        x1         x2          y
##    <fctr>      <dbl>      <dbl>
## 1       A 0.49213041 -1.27121132
```

```
## 2       A 0.63293124  0.23921944
## 3       A 0.97968782 -0.07533653
## 4       A 0.67012941  1.53051263
## 5       A 0.24507554 -0.78669550
## 6       A 0.91145207 -0.08389763
## 7       A 0.99636413 -0.33908229
## 8       A 0.06490943 -2.25764454
## 9       A 0.58626447 -0.87262047
## 10      A 0.27888727 -0.25416915
## # ... with 20 more rows
```

Notice that in this simple simulated tibble, we have coerced `x1` to be a factor variable, although this is not strictly necessary since the variable took values `A`, `B`, and `C`. When using `lm()`, even if not a factor, `R` would have treated `x1` as such. Coercion to factor is more important if a cateogical variable is coded for example as `1`, `2` and `3`. Otherwise it is treated as numeric, which creates a difference in the regression model.

The following two models illustrate the effect of factor variables on linear models.

```
cat_pred_mod_add = lm(y ~ x1 + x2, data = cat_pred)
coef(cat_pred_mod_add)
```

```
## (Intercept)         x1B         x1C          x2
##  -0.6513132   0.3670270   1.0526414   0.3998419
```

```
cat_pred_mod_int = lm(y ~ x1 * x2, data = cat_pred)
coef(cat_pred_mod_int)
```

```
## (Intercept)         x1B         x1C          x2      x1B:x2      x1C:x2
##   -1.415854    1.137031    2.715966    1.705002   -1.315671   -2.648408
```

4.6.2 Polynomials

Polynomial terms can be specified using the inhibit function `I()` or through the `poly()` function. Note that these two methods produce different coefficients, but the same residuals! This is due to the `poly()` function using orthogonal polynomials by default.

```
mod_5 = lm(Sales ~ TV + I(TV ^ 2), data = Advertising)
coef(mod_5)
```

```
##   (Intercept)            TV       I(TV^2)
##  6.114120e+00  6.726593e-02 -6.846934e-05
```

```
mod_6 = lm(Sales ~ poly(TV, degree = 2), data = Advertising)
coef(mod_6)
```

```
##           (Intercept) poly(TV, degree = 2)1 poly(TV, degree = 2)2
##             14.022500             57.572721             -6.228802
```

```
all.equal(resid(mod_5), resid(mod_6))
```

```
## [1] TRUE
```

Polynomials and interactions can be mixed to create even more complex models.

```
mod_7 = lm(Sales ~ . ^ 2 + poly(TV, degree = 3), data = Advertising)
# mod_7 = lm(Sales ~ . ^ 2 + I(TV ^ 2) + I(TV ^ 3), data = Advertising)
coef(mod_7)
```

```
##           (Intercept)                    TV                 Radio
##          6.206394e+00          2.092726e-02          3.766579e-02
##             Newspaper poly(TV, degree = 3)1 poly(TV, degree = 3)2
##          1.405289e-02                    NA         -9.925605e+00
## poly(TV, degree = 3)3              TV:Radio           TV:Newspaper
##          5.309590e+00          1.082074e-03         -5.690107e-05
##       Radio:Newspaper
##         -9.924992e-05
```

Notice here that `R` ignores the first order term from `poly(TV, degree = 3)` as it is already in the model. We could consider using the commented line instead.

4.6.3 Transformations

Note that we could also create more complex models, which allow for non-linearity, using transformations. Be aware, when doing so to the response variable, that this will affect the units of said variable. You may need to un-transform to compare to non-transformed models.

```
mod_8 = lm(log(Sales) ~ ., data = Advertising)
sqrt(mean(resid(mod_8) ^ 2)) # incorrect RMSE for Model 8
```

```
## [1] 0.1849483
```

```
sqrt(mean(resid(mod_7) ^ 2)) # RMSE for Model 7
```

```
## [1] 0.4813215
```

```
sqrt(mean(exp(resid(mod_8)) ^ 2)) # correct RMSE for Model 8
```

```
## [1] 1.023205
```

Chapter 5

Regression for Statistical Learning

When using linear models in the past, we often emphasized distributional results, which were useful for creating and performing hypothesis tests. Frequently, when developing a linear regression model, part of our goal was to **explain** a relationship.

Now, we will ignore much of what we have learned and instead simply use regression as a tool to **predict**. Instead of a model which explains relationships, we seek a model which minimizes errors.

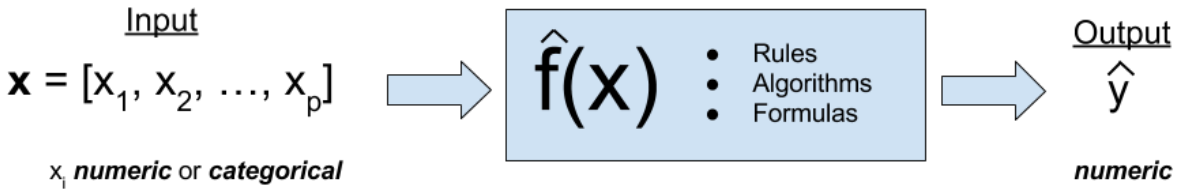


Figure 5.1:

First, note that a linear model is one of many methods used in regression. **Regression** is a form of **supervised learning**. Supervised learning deals with problems where there are both an input and an output. Regression problems are the subset of supervised learning problems with a **numeric** output.

Often one of the biggest differences between *statistical learning*, *machine learning*, *artificial intelligence* are the names used to describe variables and methods.

- The **input** can be called: input vector, feature vector, or predictors. The elements of these would be an input, feature, or predictor. The individual features can be either numeric or categorical.
- The **output** may be called: output, response, outcome, or target. The response must be numeric.

As an aside, some textbooks and statisticians use the terms independent and dependent variables to describe the response and the predictors. However, this practice can be confusing as those terms have specific meanings in probability theory.

Our goal is to find a rule, algorithm, or function which takes as input a feature vector, and outputs a response which is as close to the true value as possible. We often write the true, unknown relationship between the input and output $f(\mathbf{x})$. The relationship we learn, based on data, is written $\hat{f}(\mathbf{x})$.

From a statistical learning point-of-view, we write,

$$Y = f(\mathbf{x}) + \epsilon$$

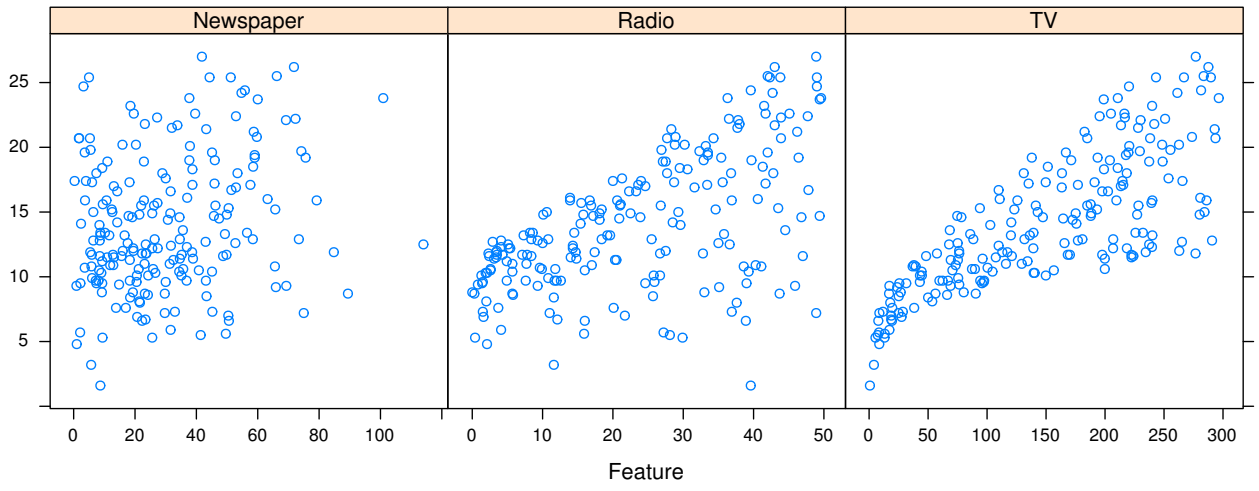
to indicate that the true response is a function of both the unknown relationship, as well as some unlearnable noise.

To discuss linear models in the context of prediction, we return to the `Advertising` data from the previous chapter.

```
Advertising
```

```
## # A tibble: 200 × 4
##       TV Radio Newspaper Sales
##    <dbl> <dbl>     <dbl> <dbl>
## 1  230.1  37.8      69.2  22.1
## 2   44.5  39.3      45.1  10.4
## 3   17.2  45.9      69.3   9.3
## 4  151.5  41.3      58.5  18.5
## 5  180.8  10.8      58.4  12.9
## 6    8.7  48.9      75.0   7.2
## 7   57.5  32.8      23.5  11.8
## 8  120.2  19.6      11.6  13.2
## 9    8.6   2.1       1.0   4.8
## 10 199.8   2.6      21.2  10.6
## # ... with 190 more rows
```

```
library(caret)
featurePlot(x = Advertising[ , c("TV", "Radio", "Newspaper")], y = Advertising$Sales)
```



5.1 Assesing Model Accuracy

There are many metrics to assess the accuracy of a regression model. Most of these measure in some way the average error that the model makes. The metric that we will be most interested in is the root-mean-square error.

$$\text{RMSE}(\hat{f}, \text{Data}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(\mathbf{x_i})\right)^2}$$

While for the sake of comparing models, the choice between RMSE and MSE is arbitrary, we have a preference for RMSE, as it has the same units as the response variable. Also, notice that in the prediction context MSE refers to an average, whereas in an ANOVA context, the denominator for MSE may not be n.

For a linear model , the estimate of f, $\hat{f}$, is given by the fitted regression line.

$$\hat{y}_i = \hat{f}(\mathbf{x_i})$$

We can write an `R` function that will be useful for performing this calculation.

```
rmse = function(actual, predicted) {
  sqrt(mean((actual - predicted) ^ 2))
}
```

5.2 Model Complexity

Aside from how well a model predicts, we will also be very interested in the complexity (flexibility) of a model. For now, we will only considered nested linear models for simplicity. Then in that case, the more predictors that a model has, the more complex the model. For the sake of assigning a numerical value to the complexity of a linear model, we will use the number of predictors, p.

We write a simple `R` function to extract this information from a model.

```
get_complexity = function(model) {
  length(coef(model)) - 1
}
```

5.3 Test-Train Split

There is an issue with fitting a model to all available data then using RMSE to determine how well the model predicts. It is essentially cheating! As a linear model becomes more complex, the RSS, thus RMSE, can never go up. It will only go down, or in very specific cases, stay the same.

This would suggest that to predict well, we should use the largest possible model! However, in reality we have hard fit to a specific dataset, but as soon as we see new data, a large model may in fact predict poorly. This is called **overfitting**.

Frequently we will take a dataset of interest and split it in two. One part of the datasets will be used to fit (train) a model, which we will call the **training** data. The remainder of the original data will be used to assess how well the model is predicting, which we will call the **test** data. Test data should *never* be used to train a model.

Note that sometimes the terms *evaluation set* and *test set* are used interchangeably. We will give somewhat specific definitions to these later. For now we will simply use a single test set for a training set.

Here we use the `sample()` function to obtain a random sample of the rows of the original data. We then use those row numbers (and remaining row numbers) to split the data accordingly. Notice we used the `set.seed()` function to allow use to reproduce the same random split each time we perform this analysis.

```
set.seed(9)
num_obs = nrow(Advertising)

train_index = sample(num_obs, size = trunc(0.50 * num_obs))
train_data = Advertising[train_index, ]
test_data = Advertising[-train_index, ]
```

We will look at two measures that assess how well a model is predicting, the **train RMSE** and the **test RMSE**.

$$\text{RMSE}_{\text{Train}} = \text{RMSE}(\hat{f}, \text{Train Data}) = \sqrt{\frac{1}{n_{\text{Tr}}} \sum_{i \in \text{Train}} \left(y_i - \hat{f}(\mathbf{x_i})\right)^2}$$

Here n_{Tr} is the number of observations in the train set. Train RMSE will still always go down (or stay the same) as the complexity of a linear model increases. That means train RMSE will not be useful for comparing models, but checking that it decreases is a useful sanity check.

$$\text{RMSE}_{\text{Test}} = \text{RMSE}(\hat{f}, \text{Train Data}) = \sqrt{\frac{1}{n_{\text{Te}}} \sum_{i \in \text{Test}} \left(y_i - \hat{f}(\mathbf{x_i})\right)^2}$$

Here n_{Te} is the number of observations in the test set. Test RMSE uses the model fit to the training data, but evaluated on the unused test data. This is a measure of how well the fitted model will predict **in general**, not simply how well it fits data used to train the model, as is the case with train RMSE. What happens to test RMSE as the size of the model increases? That is what we will investigate.

We will start with the simplest possible linear model, that is, a model with no predictors.

```
fit_0 = lm(Sales ~ 1, data = train_data)
get_complexity(fit_0)
```

```
## [1] 0
```

```
# train RMSE
sqrt(mean((train_data$Sales - predict(fit_0, train_data)) ^ 2))
```

```
## [1] 4.788513
```

```
# test RMSE
sqrt(mean((test_data$Sales - predict(fit_0, test_data)) ^ 2))
```

```
## [1] 5.643574
```

The previous two operations obtain the train and test RMSE. Since these are operations we are about to use repeatedly, we should use the function that we happen to have already written.

```
# train RMSE
rmse(actual = train_data$Sales, predicted = predict(fit_0, train_data))
```

```
## [1] 4.788513
```

```
# test RMSE
rmse(actual = test_data$Sales, predicted = predict(fit_0, test_data))
```

```
## [1] 5.643574
```

This function can actually be improved for the inputs that we are using. We would like to obtain train and test RMSE for a fitted model, given a train or test dataset, and the appropriate response variable.

```
get_rmse = function(model, data, response) {
  rmse(actual = data[, response],
       predicted = predict(model, data))
}
```

By using this function, our code becomes easier to read, and it is more obvious what task we are accomplishing.

```
get_rmse(model = fit_0, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 4.788513
```

```
get_rmse(model = fit_0, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 5.643574
```

5.4 Adding Flexibility to Linear Models

Each successive model we fit will be more and more flexible using both interactions and polynomial terms. We will see the training error decrease each time the model is made more flexible. We expect the test error to decrease a number of times, then eventually start going up, as a result of overfitting.

```
fit_1 = lm(Sales ~ ., data = train_data)
get_complexity(fit_1)
```

```
## [1] 3
```

```
get_rmse(model = fit_1, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 1.637699
```

```
get_rmse(model = fit_1, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 1.737574
```

```
fit_2 = lm(Sales ~ Radio * Newspaper * TV, data = train_data)
get_complexity(fit_2)
```

```
## [1] 7
```

```
get_rmse(model = fit_2, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 0.7797226
```

```
get_rmse(model = fit_2, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 1.110372
```

```
fit_3 = lm(Sales ~ Radio * Newspaper * TV + I(TV ^ 2), data = train_data)
get_complexity(fit_3)
```

```
## [1] 8
```

```
get_rmse(model = fit_3, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 0.4960149
```

```
get_rmse(model = fit_3, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 0.7320758
```

```
fit_4 = lm(Sales ~ Radio * Newspaper * TV +
           I(TV ^ 2) + I(Radio ^ 2) + I(Newspaper ^ 2), data = train_data)
get_complexity(fit_4)
```

```
## [1] 10
```

```
get_rmse(model = fit_4, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 0.488771
```

```
get_rmse(model = fit_4, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 0.7466312
```

```
fit_5 = lm(Sales ~ Radio * Newspaper * TV +
           I(TV ^ 2) * I(Radio ^ 2) * I(Newspaper ^ 2), data = train_data)
get_complexity(fit_5)
```

```
## [1] 14
```

```
get_rmse(model = fit_5, data = train_data, response = "Sales") # train RMSE
```

```
## [1] 0.4705201
```

```
get_rmse(model = fit_5, data = test_data, response = "Sales") # test RMSE
```

```
## [1] 0.8425384
```

5.5 Choosing a Model

To better understand the relationship between train RMSE, test RMSE, and model complexity, we summarize our results, as the above is somewhat cluttered.

First, we recap the models that we have fit.

```
fit_1 = lm(Sales ~ ., data = train_data)
fit_2 = lm(Sales ~ Radio * Newspaper * TV, data = train_data)
fit_3 = lm(Sales ~ Radio * Newspaper * TV + I(TV ^ 2), data = train_data)
fit_4 = lm(Sales ~ Radio * Newspaper * TV +
           I(TV ^ 2) + I(Radio ^ 2) + I(Newspaper ^ 2), data = train_data)
fit_5 = lm(Sales ~ Radio * Newspaper * TV +
           I(TV ^ 2) * I(Radio ^ 2) * I(Newspaper ^ 2), data = train_data)
```

Next, we create a list of the models fit.

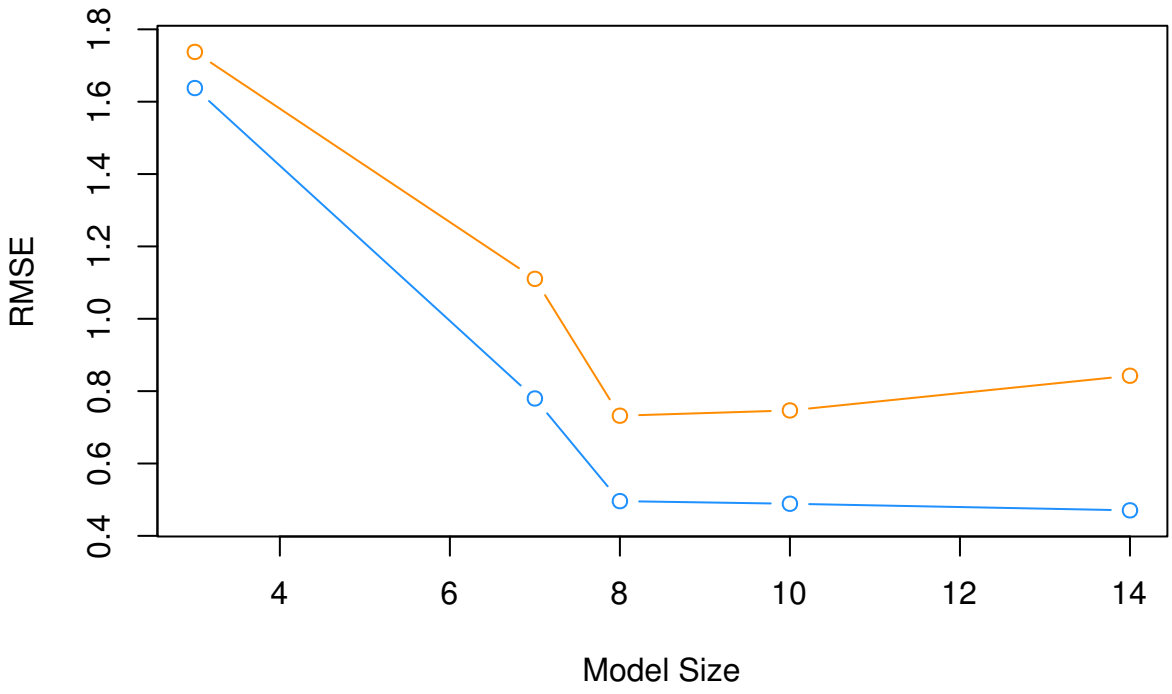
```
model_list = list(fit_1, fit_2, fit_3, fit_4, fit_5)
```

We then obtain train RMSE, test RMSE, and model complexity for each.

```
train_rmse = sapply(model_list, get_rmse, data = train_data, response = "Sales")
test_rmse = sapply(model_list, get_rmse, data = test_data, response = "Sales")
model_complexity = sapply(model_list, get_complexity)
```

We then plot the results. The train RMSE can be seen in blue, while the test RMSE is given in orange.

```
plot(model_complexity, train_rmse, type = "b",
     ylim = c(min(c(train_rmse, test_rmse)) - 0.02,
              max(c(train_rmse, test_rmse)) + 0.02),
     col = "dodgerblue",
     xlab = "Model Size",
     ylab = "RMSE")
lines(model_complexity, test_rmse, type = "b", col = "darkorange")
```



We also summarize the results as a table. `fit_1` is the least flexible, and `fit_5` is the most flexible. We see the Train RMSE decrease as flexibility increases. We see that the Test RMSE is smallest for `fit_3`, thus is the model we believe will perform the best on future data not used to train the model. Note this may not be the best model, but it is the best model of the models we have seen in this example.

Model	Train RMSE	Test RMSE	Predictors
`fit_1`	1.6376991	1.7375736	3
`fit_2`	0.7797226	1.1103716	7
`fit_3`	0.4960149	0.7320758	8
`fit_4`	0.488771	0.7466312	10
`fit_5`	0.4705201	0.8425384	14

To summarize:

- **Underfitting models:** In general *High* Train RMSE, *High* Test RMSE. Seen in `fit_1` and `fit_2`.
- **Overfitting models:** In general *Low* Train RMSE, *High* Test RMSE. Seen in `fit_4` and `fit_5`.

Specifically, we say that a model is overfitting if there exists a less complex model with lower Test RMSE. Then a model is underfitting if there exists a more complex model with lower Test RMSE.

A number of notes on these results:

- The labels of under and overfitting are *relative* to the best model we see, `fit_3`. Any model more complex with higher Test RMSE is overfitting. Any model less complex with higher Test RMSE is underfitting.
- The train RMSE is guaranteed to follow this non-increasing pattern. The same is not true of test RMSE. Here we see a nice U-shaped curve. There are theoretical reasons why we should expect this, but that is on average. Because of the randomness of one test-train split, we may not always see this result. Re-perform this analysis with a different seed value and the pattern may not hold. We will discuss why we expect this next chapter. We will discuss how we can help create this U-shape much later.

- Often we expect train RMSE to be lower than test RMSE. Again, due to the randomness of the split, you may get lucky and this will not be true.

A final note on the analysis performed here; we paid no attention whatsoever to the "assumptions" of a linear model. We only sought a model that **predicted** well, and paid no attention to a model for **explaination**. Hypothesis testing did not play a role in deciding the model, only prediction accuracy. Collinearity? We don't care. Assumptions? Still don't care. Diagnostics? Never heard of them. (These statements are a little over the top, and not completely true, but just to drive home the point that we only care about prediction. Often we latch onto methods that we have seen before, even when they are not needed.)

Chapter 6

Simulating the Bias–Variance Tradeoff

Consider the general regression setup

$$y = f(\mathbf{x}) + \epsilon$$

with

$$E[\epsilon] = 0 \quad \text{and} \quad \text{var}(\epsilon) = \sigma^2.$$

6.1 Bias-Variance Decomposition

Using $\hat{f}(\mathbf{x})$, trained with data, to estimate $f(\mathbf{x})$, we are interested in the expected prediction error. Specifically, considered making a prediction of $y_0 = f(\mathbf{x}_0) + \epsilon$ at the point $\mathbf{x}_0$.

In that case, we have

$$E\left[\left(y_0 - \hat{f}(\mathbf{x}_0)\right)^2\right] = \text{bias}\left(\hat{f}(\mathbf{x}_0)\right)^2 + \text{var}\left(\hat{f}(\mathbf{x}_0)\right) + \sigma^2.$$

Recall the definition of the bias of an estimate.

$$\text{bias}\left(\hat{f}(\mathbf{x}_0)\right) = E\left[\hat{f}(\mathbf{x}_0)\right] - f(\mathbf{x}_0)$$

So, we have decomposed the error into two types; **reducible** and **irreducible**. The reducible can be further decomposed into the squared **bias** and **variance** of the estimate. We can "control" these through our choice of model. The irreducible, is noise, that should not and cannot be modeled.

6.2 Simulation

We will illustrate this decomposition, and the resulting bias-variance tradeoff through simulation. Suppose we would like a train a model to learn the function $f(x) = x^2$.

```
f = function(x) {
  x ^ 2
}
```

More specifically,

$$y = x^2 + \epsilon$$

where

$$\epsilon \sim N(\mu = 0, \sigma^2 = 0.3^2).$$

We write a function which generates data accordingly.

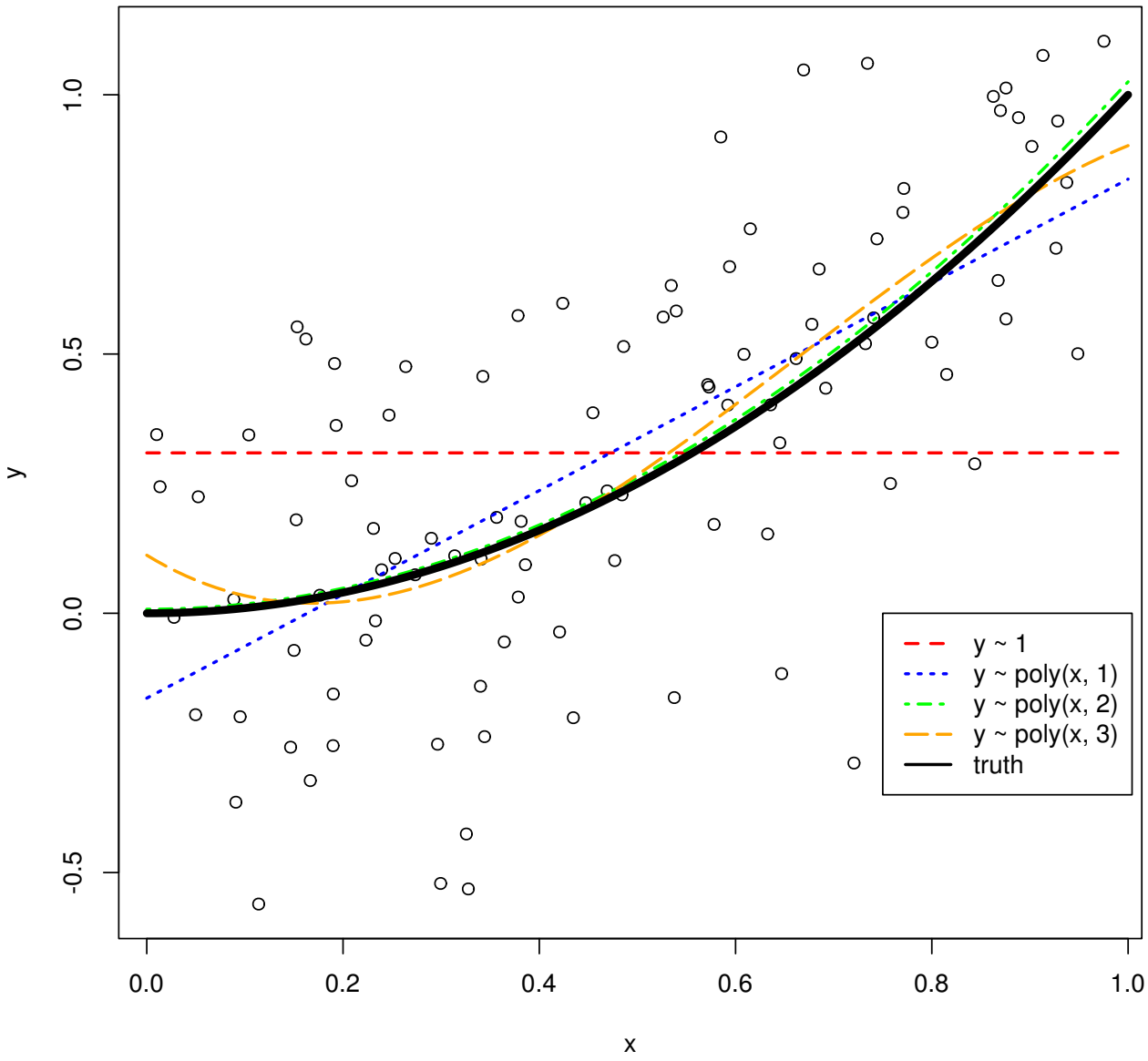
```
get_sim_data = function(f, sample_size = 100) {
  x = runif(n = sample_size, min = 0, max = 1)
  y = f(x) + rnorm(n = sample_size, mean = 0, sd = 0.3)
  data.frame(x, y)
}
```

To get a sense of the data, we generate one simulated dataset, and fit the four models that we will be of interest.

```
sim_data = get_sim_data(f, sample_size = 100)

fit_1 = lm(y ~ 1, data = sim_data)
fit_2 = lm(y ~ poly(x, degree = 1), data = sim_data)
fit_3 = lm(y ~ poly(x, degree = 2), data = sim_data)
fit_4 = lm(y ~ poly(x, degree = 3), data = sim_data)
```

Plotting these four trained models, we see that the zero predictor model (red) does very poorly. The single predictor model (blue) is reasonable, but we can see that the two (green) and three (orange) predictor models seem more appropriate. Between these latter two, it is hard to see which seems more appropriate.

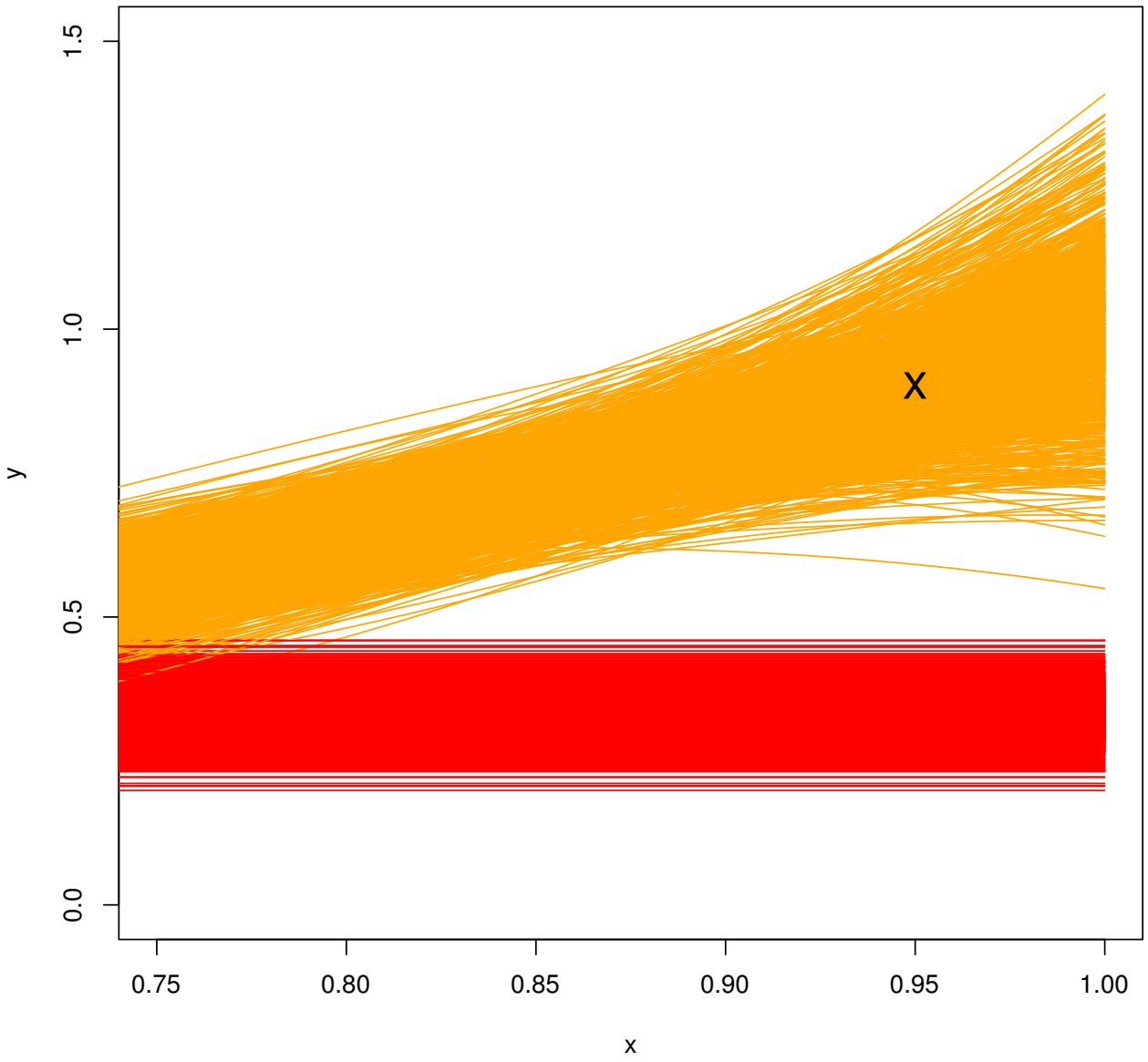
```
set.seed(430)
plot(y ~ x, data = sim_data)
grid = seq(from = 0, to = 1, by = 0.01)
lines(grid, predict(fit_1, newdata = data.frame(x = grid)),
      col = "red", lwd = 2, lty = 2)
lines(grid, predict(fit_2, newdata = data.frame(x = grid)),
      col = "blue", lwd = 2, lty = 3)
lines(grid, predict(fit_3, newdata = data.frame(x = grid)),
      col = "green", lwd = 2, lty = 4)
lines(grid, predict(fit_4, newdata = data.frame(x = grid)),
      col = "orange", lwd = 2, lty = 5)
lines(grid, f(grid), col = "black", lwd = 5)
legend(x = 0.75, y = 0,
       c("y ~ 1", "y ~ poly(x, 1)", "y ~ poly(x, 2)",  "y ~ poly(x, 3)", "truth"),
       col = c("red", "blue", "green", "orange", "black"), lty = c(2, 3, 4, 5, 1), lwd = 2)
```



We will now use simulation to estimate the bias, variance, and mean squared error for the estimates for $f(x)$ given by these models at the point $x_0 = 0.95$. We use simulation to complete this task, as performing the exact calculations are always difficult, and often impossible.

```
set.seed(1)
n_sims = 1000
n_models = 4
x0 = 0.95
predictions = matrix(0, nrow = n_sims, ncol = n_models)
sim_data = get_sim_data(f, sample_size = 100)
plot(y ~ x, data = sim_data, col = "white", xlim = c(0.75, 1), ylim = c(0, 1.5))

for (i in 1:n_sims) {

  sim_data = get_sim_data(f, sample_size = 100)

  fit_1 = lm(y ~ 1, data = sim_data)
```

```
  fit_2 = lm(y ~ poly(x, degree = 1), data = sim_data)
  fit_3 = lm(y ~ poly(x, degree = 2), data = sim_data)
  fit_4 = lm(y ~ poly(x, degree = 3), data = sim_data)

  lines(grid, predict(fit_1, newdata = data.frame(x = grid)), col = "red", lwd = 1)
  # lines(grid, predict(fit_2, newdata = data.frame(x = grid)), col = "blue", lwd = 1)
  # lines(grid, predict(fit_3, newdata = data.frame(x = grid)), col = "green", lwd = 1)
  lines(grid, predict(fit_4, newdata = data.frame(x = grid)), col = "orange", lwd = 1)

  predictions[i, ] <- c(
    predict(fit_1, newdata = data.frame(x = x0)),
    predict(fit_2, newdata = data.frame(x = x0)),
    predict(fit_3, newdata = data.frame(x = x0)),
    predict(fit_4, newdata = data.frame(x = x0))
  )
}

points(x0, f(x0), col = "black", pch = "x", cex = 2)
```



The above plot shows the 1000 trained models for each of the zero predictor and three predictor models. (We have exlcuded the one and two predictor models for clarity of the plot.) The truth at $x_0 = 0.95$ is given by a black "X". We see that the red lines for the zero predictor model are on average wrong, with some variability. The orange lines for the three predictor model are on average correct, but with more variance.

6.3 Bias-Variance Tradeoff

To evaluate the bias and variance, we simulate values for the response y at $x_0 = 0.95$ according to the true model.

```
eps = rnorm(n = n_sims, mean = 0, sd = 0.3)
y0 = f(x0) + eps
```

`R` already has a function to calculate variance, however, we add functions for bias and mean squared error.

```
get_bias = function(estimate, truth) {
  mean(estimate) - truth
}

get_mse = function(estimate, truth) {
  mean((estimate - truth) ^ 2)
}
```

When then use the predictions obtained from the above simulation to estimate the bias, variance and mean squared error for estimating $f(x)$ at $x_0 = 0.95$ for the four models.

```
bias = apply(predictions, 2, get_bias, f(x0))
variance = apply(predictions, 2, var)
mse = apply(predictions, 2, get_mse, y0)
```

We summarize these results in the following table.

Model	Squared Bias	Variance (Of Estimate)	MSE
`fit_1`	0.322916	0.001784	0.4201411
`fit_2`	0.0136794	0.0036355	0.1145159
`fit_3`	0.0000036	0.0058178	0.1031294
`fit_4`	0.0000009	0.0079906	0.1053599

A number of things to notice here:

- We use squared bias in this table. Since bias can be positive or negative, squared bias is more useful for observing the trend as complexity increases.
- The squared bias trend which we see here is **decreasing** bias as complexity increases, which we expect to see in general.
- The exact opposite is true of variance. As model complexity increases, variance **increases**.
- The mean squared error, which is a function of the bias and variance, decreases, then increases. This is a result of the bias-variance tradeoff. We can decrease bias, by increases variance. Or, we can decrease variance by increasing bias. By striking the correct balance, we can find a good mean squared error.

We can check for these trends with the `diff()` function in `R`.

```
all(diff(bias ^ 2) < 0)
```

```
## [1] TRUE
```

```
all(diff(variance) > 0)
```

```
## [1] TRUE
```

```
diff(mse)
```

```
## [1] -0.305625170 -0.011386537  0.002230515
```

Notice that the table lacks a column for the variance of the noise. Add this to squared bias and variance would give the mean squared error. However, notice that we are simulation to estiamte the bias and variance, so the relationship is not exact. If we used more replications of the simulation, these two values would move closer together.

```
bias ^ 2 + variance + var(eps)
```

```
## [1] 0.4209744 0.1135892 0.1020958 0.1042659
```

```
mse
```

```
## [1] 0.4201411 0.1145159 0.1031294 0.1053599
```

Chapter 7

Classification

Classification is a form of **supervised learning** where the response variable is categorical, as opposed to numeric for regression. *Our goal is to find a rule, algorithm, or function which takes as input a feature vector, and outputs a category which is the true category as often as possible.*

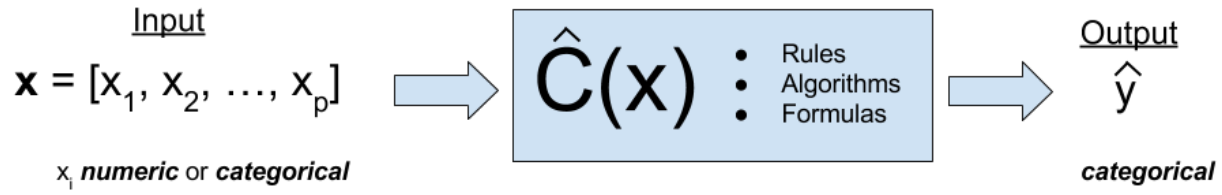


Figure 7.1:

That is, the classifier $\hat{C}$ returns the predicted category $\hat{y}$.

$$\hat{y}_i = \hat{C}(\mathbf{x_i})$$

To build our first classifier, we will use the `Default` dataset from the `ISLR` package.

```
library(ISLR)
library(tibble)
as_tibble(Default)
```

```
## # A tibble: 10,000 × 4
##    default student   balance    income
##     <fctr>  <fctr>     <dbl>     <dbl>
## 1       No      No  729.5265 44361.625
## 2       No     Yes  817.1804 12106.135
## 3       No      No 1073.5492 31767.139
## 4       No      No  529.2506 35704.494
## 5       No      No  785.6559 38463.496
## 6       No     Yes  919.5885  7491.559
## 7       No      No  825.5133 24905.227
## 8       No     Yes  808.6675 17600.451
```

```
## 9       No      No 1161.0579 37468.529
## 10      No      No    0.0000 29275.268
## # ... with 9,990 more rows
```

Our goal is to properly classify individuals as defaulters based on student status, credit card balance, and income. Be aware that the response `default` is a factor, as is the predictor `student`.

```
is.factor(Default$default)
```

```
## [1] TRUE
```

```
is.factor(Default$student)
```

```
## [1] TRUE
```

As we did with regression, we test-train split our data. In this case, using 50% for each.

```
set.seed(42)
train_index = sample(nrow(Default), 5000)
train_default = Default[train_index, ]
test_default = Default[-train_index, ]
```

7.1 Visualization for Classification

Often, some simple visualizations can suggest simple classification rules. To quickly create some useful visualizations, we use the `featurePlot()` function from the `caret()` package.

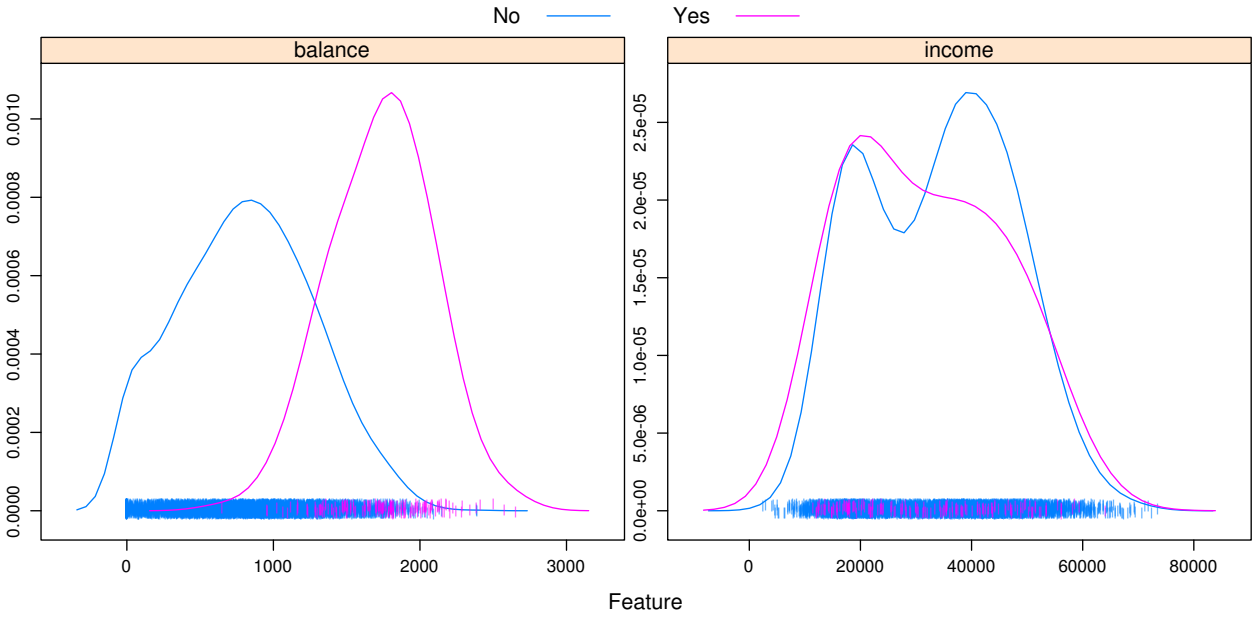
```
library(caret)
```

A density plot can often suggest a simple split based on a numeric predictor. Essentially this plot graphs a density estimate

$$f_{X_i}(x_i \mid y = k)$$

for each numeric predictor x_i and each category k of the response y.

```
featurePlot(x = train_default[, c("balance", "income")],
            y = train_default$default,
            plot = "density",
            scales = list(x = list(relation = "free"),
                          y = list(relation = "free")),
            adjust = 1.5,
            pch = "|",
            layout = c(2, 1),
            auto.key = list(columns = 2))
```

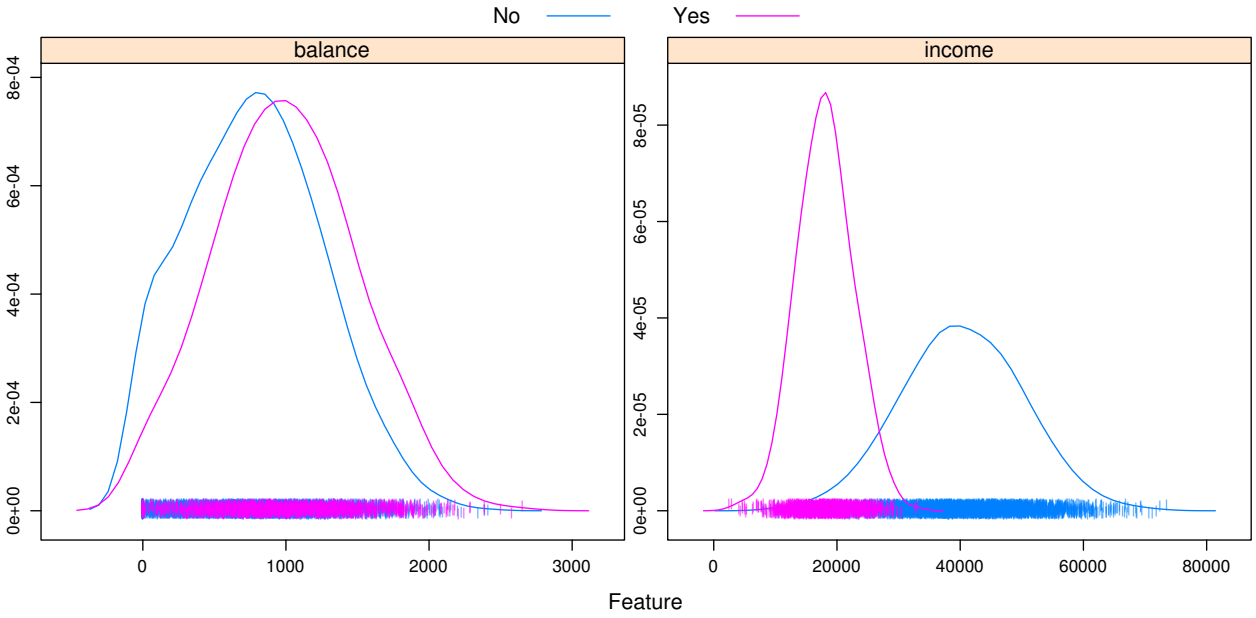


Some notes about the arguments to this function:

- `x` is a data frame containing only **numeric predictors**. It would be nonsensical to estimate a density for a categorical predictor.
- `y` is the response variable. It needs to be a factor variable. If coded as `0` and `1`, you will need to coerce to factor for plotting.
- `plot` specifies the type of plot, here `density`.
- `scales` defines the scale of the axes for each plot. By default, the axis of each plot would be the same, which often is not useful, so the arguments here, a different axis for each plot, will almost always be used.
- `adjust` specifies the amount of smoothing used for the density estimate.
- `pch` specifies the **p**lot **ch**aracter used for the bottom of the plot.
- `layout` places the individual plots into rows and columns. For some odd reason, it is given as (col, row).
- `auto.key` defines the key at the top of the plot. The number of columns should be the number of categories.

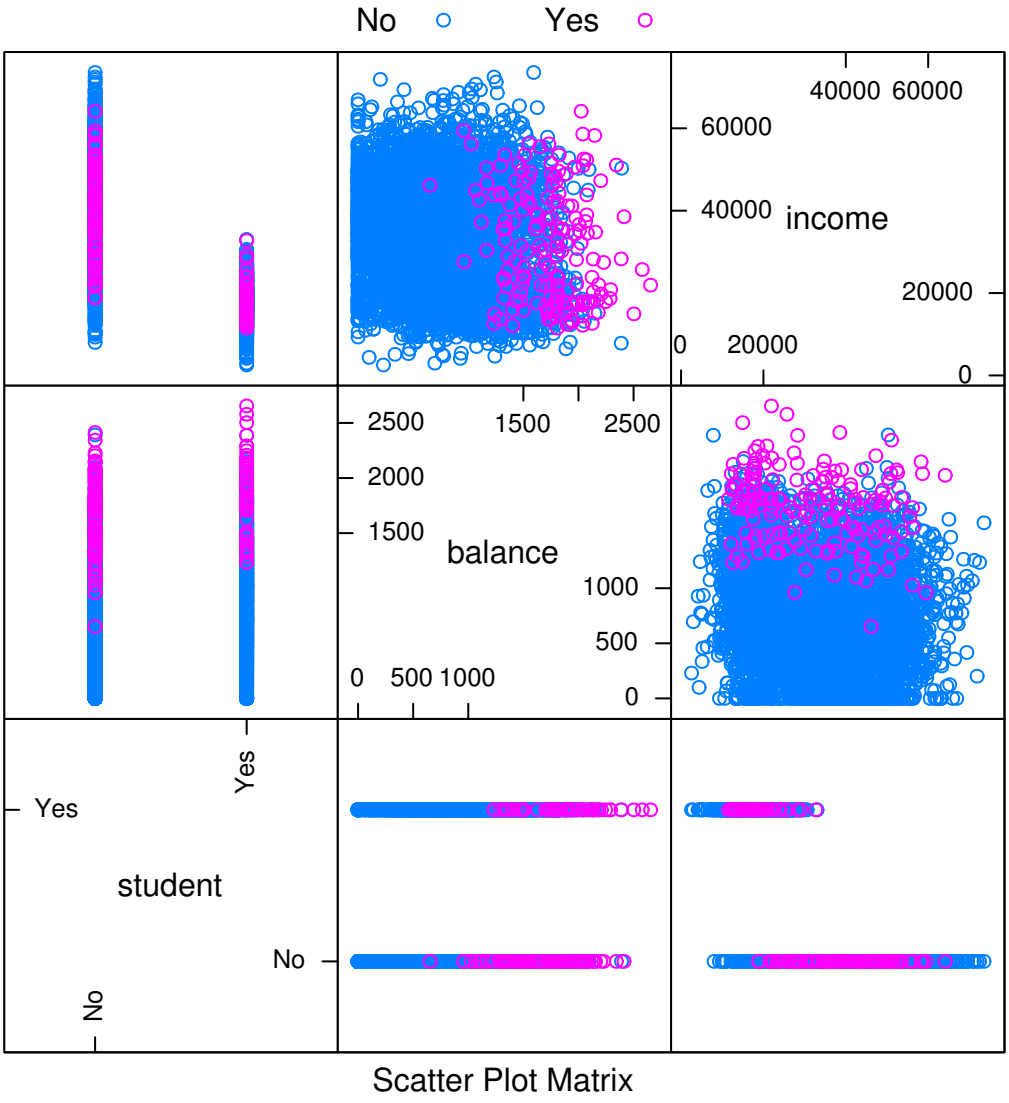
It seems that the income variable by itself is not particularly useful. However, there seems to be a big difference in default status at a `balance` of about 1400. We will use this information shortly.

```
featurePlot(x = train_default[, c("balance", "income")],
            y = train_default$student,
            plot = "density",
            scales = list(x = list(relation = "free"),
                          y = list(relation = "free")),
            adjust = 1.5,
            pch = "|",
            layout = c(2, 1),
            auto.key = list(columns = 2))
```



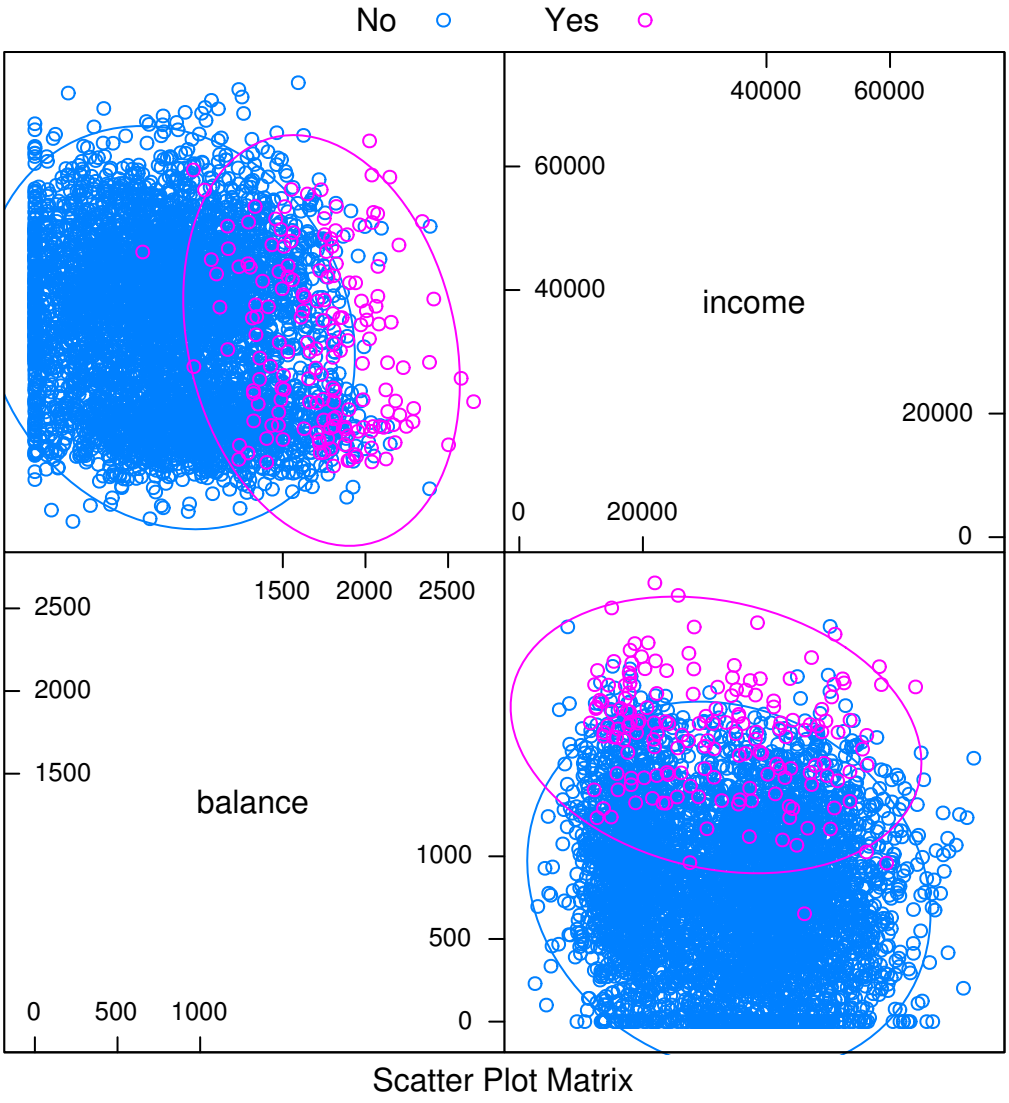
Above, we create a similar plot, except with `student` as the response. We see that students often carry a slightly larger balance, and have far lower income. This will be useful to know when making more complicated classifiers.

```
featurePlot(x = train_default[, c("student", "balance", "income")],
            y = train_default$default,
            plot = "pairs",
            auto.key = list(columns = 2))
```



We can use `plot = "pairs"` to consider multiple variables at the same time. This plot reinforces using `balance` to create a classifier, and again shows that `income` seems not that useful.

```
library(ellipse)
featurePlot(x = train_default[, c("balance", "income")],
            y = train_default$default,
            plot = "ellipse",
            auto.key = list(columns = 2))
```



Scatter Plot Matrix

Similar to `pairs` is a plot of type `ellipse`, which requires the `ellipse` package. Here we only use numeric predictors, as essentially we are assuming multivariate normality. The ellipses mark points of equal density. This will be useful later when discussing LDA and QDA.

7.2 A Simple Classifier

A very simple classifier is a rule based on a cutoff c for a particular input variable x.

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & x > c \\ 0 & x \leq c \end{cases}$$

Based on the first plot, we believe we can use `balance` to create a reasonable classifier. In particular,

$$\hat{C}(\text{balance}) = \begin{cases} \text{Yes} & \text{balance} > 1400 \\ \text{No} & \text{balance} \leq 1400 \end{cases}$$

So we predict an individual is a defaulter if their `balance` is above 1400, and not a defaulter if the balance is 1400 or less.

```
simple_class = function(x, cutoff, above = 1, below = 0) {
  ifelse(x > cutoff, above, below)
}
```

We write a simple `R` function that compares a variable to a cutoff, then use it to make predictions on the train and test sets with our chosen variable and cutoff.

```
train_pred = simple_class(x = train_default$balance,
                          cutoff = 1400, above = "Yes", below = "No")
test_pred = simple_class(x = test_default$balance,
                         cutoff = 1400, above = "Yes", below = "No")
head(train_pred, n = 10)
```

```
##  [1] "No"  "Yes" "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"
```

7.3 Metrics for Classification

In the classification setting, there are a large number of metrics to asses how well a classifier is performing.

One of the most obvious things to do is arrange predictions and true values in a cross table.

```
(train_tab = table(predicted = train_pred, actual = train_default$default))
```

```
##          actual
## predicted   No  Yes
##       No  4319   29
##       Yes  513  139
```

```
(test_tab = table(predicted = test_pred, actual = test_default$default))
```

```
##          actual
## predicted   No  Yes
##       No  4361   23
##       Yes  474  142
```

Often we give specific names to individual cells of these tables, and in the predictive setting, we would call this table a **confusion matrix**. Be aware, that the placement of Actual and Predicted values affects the names of the cells, and often the matrix may be presented transposed.

In statistics, we label the errors Type I and Type II, but these are hard to remember. False Positive and False Negative are more descriptive, so we choose to use these.

The `confusionMatrix()` function from the `caret` package can be used to obtain a wealth of additional information, which we see output below for the test data. Note that we specify which category is considered "positive."

		Actual	
		False (0)	**True** (1)
Predicted	**False** (0)	True Negative (**TN**)	False Negative (**FN**)
	True (1)	False Positive (**FP**)	True Positive (**TP**)

Figure 7.2:

```
train_con_mat = confusionMatrix(train_tab, positive = "Yes")
(test_con_mat = confusionMatrix(test_tab, positive = "Yes"))
```

```
## Confusion Matrix and Statistics
##
##          actual
## predicted   No  Yes
##       No  4361   23
##       Yes  474  142
##
##                Accuracy : 0.9006
##                  95% CI : (0.892, 0.9088)
##     No Information Rate : 0.967
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.3287
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.8606
##             Specificity : 0.9020
##          Pos Pred Value : 0.2305
##          Neg Pred Value : 0.9948
##              Prevalence : 0.0330
##          Detection Rate : 0.0284
##    Detection Prevalence : 0.1232
##       Balanced Accuracy : 0.8813
##
##        'Positive' Class : Yes
##
```

The most common, and most important metric is the **classification accuracy**.

$$\text{Acc}(\hat{C}, \text{Data}) = \frac{1}{n} \sum_{i=1}^{n} I(y_i = \hat{C}(\mathbf{x_i}))$$

Here, I is an indicator function, so we are essentially calculating the proportion of predicted classes that match the true class.

$$I(y_i = \hat{C}(x)) = \begin{cases} 1 & y_i = \hat{C}(x) \\ 0 & y_i \neq \hat{C}(x) \end{cases}$$

It is also common to discuss the **misclassification rate**, or classification error, which is simply one minus the accuracy.

Like regression, we often split the data, and then consider Train Accuracy and Test Accuracy. Test Accuracy will be used as a measure of how well a classifier will work on unseen future data.

$$\text{Acc}_{\text{Train}}(\hat{C}, \text{Train Data}) = \frac{1}{n_{Tr}} \sum_{i \in \text{Train}} I(y_i = \hat{C}(\mathbf{x_i}))$$

$$\text{Acc}_{\text{Test}}(\hat{C}, \text{Test Data}) = \frac{1}{n_{Te}} \sum_{i \in \text{Test}} I(y_i = \hat{C}(\mathbf{x_i}))$$

These accuracy values are given by calling `confusionMatrix()`, or, if stored, can be accessed directly.

```
train_con_mat$overall["Accuracy"]
```

```
## Accuracy
##   0.8916
```

```
test_con_mat$overall["Accuracy"]
```

```
## Accuracy
##   0.9006
```

Sometimes guarding against making certain errors, FP or FN, are more important than simply finding the best accuracy. Thus, sometimes we will consider **sensitivity** and **specificity**.

$$\text{Sens} = \text{True Positive Rate} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

```
test_con_mat$byClass["Sensitivity"]
```

```
## Sensitivity
##   0.8606061
```

$$\text{Spec} = \text{True Negative Rate} = \frac{\text{TN}}{\text{N}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

```
test_con_mat$byClass["Specificity"]
```

```
## Specificity
##   0.9019648
```

Like accuracy, these can easily be found using `confusionMatrix()`.

When considering how well a classifier is performing, often, it is understandable to assume that any accuracy in a binary classification problem above 0.50, is a reasonable classifier. This however is not the case. We need to consider the **balance** of the classes. To do so, we look at the **prevalence** of positive cases.

$$\text{Prev} = \frac{\text{P}}{\text{Total Obs}} = \frac{\text{TP} + \text{FN}}{\text{Total Obs}}$$

```
train_con_mat$byClass["Prevalence"]
```

```
## Prevalence
##     0.0336
```

```
test_con_mat$byClass["Prevalence"]
```

```
## Prevalence
##      0.033
```

Here, we see an extremely low prevalence, which suggests an even simpler classifier than our current based on `balance`.

$$\hat{C}(\text{balance}) = \begin{cases} \text{No} & \text{balance} > 1400 \\ \text{No} & \text{balance} \leq 1400 \end{cases}$$

This classifier simply classifies all observations as negative cases.

```
pred_all_no = simple_class(test_default$balance,
                           cutoff = 1400, above = "No", below = "No")
table(predicted = pred_all_no, actual = test_default$default)
```

```
##          actual
## predicted   No  Yes
##        No 4835  165
```

The `confusionMatrix()` function won't even accept this table as input, because it isn't a full matrix, only one row, so we calculate some metrics "by hand".

```
4835 / (4835 + 165) # test accuracy
```

```
## [1] 0.967
```

```
1 - 0.0336 # 1 - (train prevelence)
```

```
## [1] 0.9664
```

```
1 - 0.033 # 1 - (test prevelence)
```

```
## [1] 0.967
```

This classifier does better than the previous. But the point is, in reality, to create a good classifier, we should obtain a test accuracy better than 0.967, which is obtained by simply manipulating the prevalence. Next chapter, we'll introduce much better classifiers which should have no problem accomplishing this task.

Chapter 8

Logistic Regression

In this chapter, we continue our discussion of classification. We introduce our first model for classification, logistic regression. To begin, we return to the `Default` dataset from the previous chapter.

```
library(ISLR)
library(tibble)
as_tibble(Default)
```

```
## # A tibble: 10,000 × 4
##    default student   balance    income
##     <fctr>  <fctr>     <dbl>     <dbl>
## 1       No      No  729.5265 44361.625
## 2       No     Yes  817.1804 12106.135
## 3       No      No 1073.5492 31767.139
## 4       No      No  529.2506 35704.494
## 5       No      No  785.6559 38463.496
## 6       No     Yes  919.5885  7491.559
## 7       No      No  825.5133 24905.227
## 8       No     Yes  808.6675 17600.451
## 9       No      No 1161.0579 37468.529
## 10      No      No    0.0000 29275.268
## # ... with 9,990 more rows
```

We also repeat the test-train split from the previous chapter.

```
set.seed(42)
default_index = sample(nrow(Default), 5000)
default_train = Default[default_index, ]
default_test = Default[-default_index, ]
```

8.1 Linear Regression

Before moving on to logistic regression, why not plain, old, linear regression?

```
default_train_lm = default_train
default_test_lm = default_test
```

Since linear regression expects a numeric response variable, we coerce the response to be numeric. (Notice that we also shift the results, as we require `0` and `1`, not `1` and `2`.) Notice we have also copied the dataset so that we can return the original data with factors later.

```
default_train_lm$default = as.numeric(default_train_lm$default) - 1
default_test_lm$default = as.numeric(default_test_lm$default) - 1
```

Why would we think this should work? Recall that,

$$\hat{E}[Y \mid X = x] = X\hat{\beta}.$$

Since Y is limited to values of 0 and 1, we have

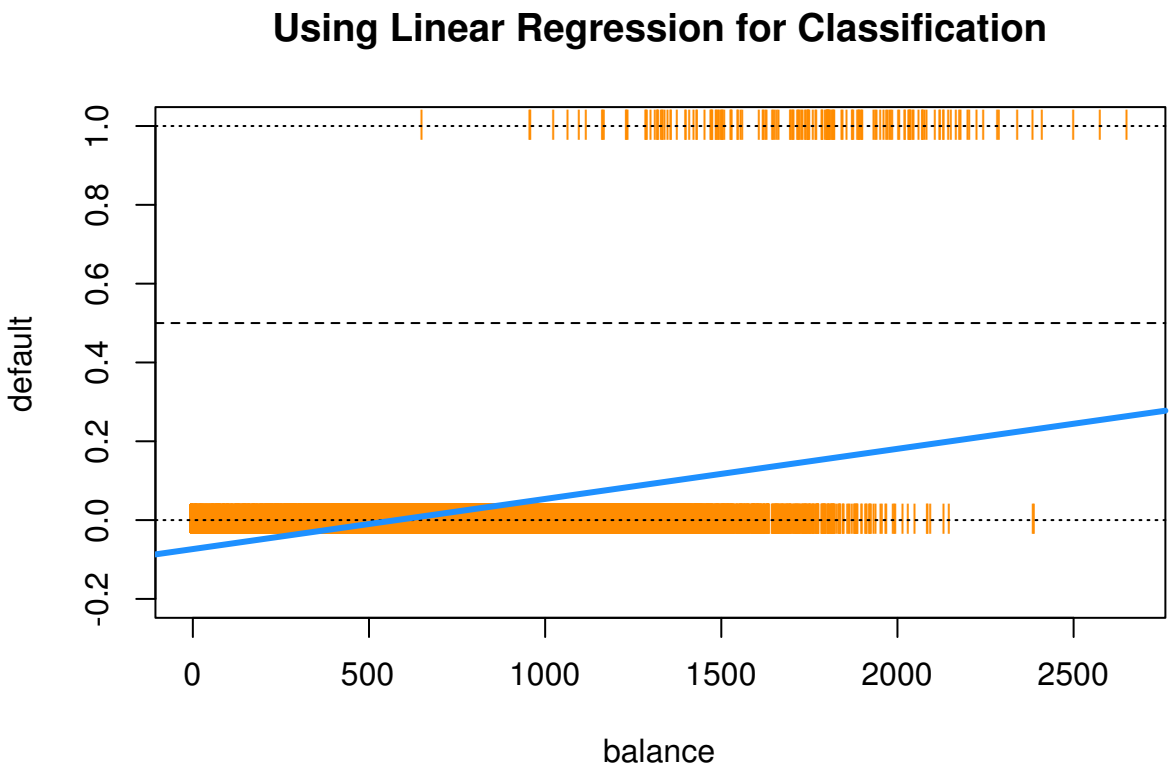
$$E[Y \mid X = x] = P[Y = 1 \mid X = x].$$

It would then seem reasonable that $X\hat{\beta}$ is a reasonable estimate of $P[Y = 1 \; X = x]$. We test this on the `Default` data.

```
model_lm = lm(default ~ balance, data = default_train_lm)
```

Everything seems to be working, until we plot the results.

```
plot(default ~ balance, data = default_train_lm,
     col = "darkorange", pch = "|", ylim = c(-0.2, 1),
     main = "Using Linear Regression for Classification")
abline(h = 0, lty = 3)
abline(h = 1, lty = 3)
abline(h = 0.5, lty = 2)
abline(model_lm, lwd = 3, col = "dodgerblue")
```



Two issues arise. First, all of the predicted probabilities are below 0.5 That means, we would classify every observation as a `"No"`. This is certainly possible, but not what we would expect.

```
all(predict(model_lm) < 0.5)
```

```
## [1] TRUE
```

The next, and bigger issue, is predicted probabilities less than 0.

```
any(predict(model_lm) < 0)
```

```
## [1] TRUE
```

8.2 Bayes Classifier

Why are we using a predicted probability of 0.5 as the cutoff for classification? Recall, the Bayes Classifier, which minimizes the classification error:

$$C^B(\mathbf{x}) = \underset{k}{\text{argmax}} \ P[Y = k \mid \mathbf{X} = \mathbf{x}]$$

So, in the binary classification problem, we will use predicted probabilities

$$\hat{p}(\mathbf{x}) = \hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}]$$

and

$$\hat{P}[Y = 0 \mid \mathbf{X} = \mathbf{x}]$$

and then classify to the larger of the two. We actually only need to consider a single probability, usually for $\hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}]$. Since we use it so often, we give it the shorthand notation, $\hat{p}(\mathbf{x})$. Then the classifier is written,

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & \hat{p}(\mathbf{x}) > 0.5 \\ 0 & \hat{p}(\mathbf{x}) \leq 0.5 \end{cases}$$

8.3 Logistic Regression with `glm()`

To better estimate the probability

$$p(\mathbf{x}) = P[Y = 1 \mid \mathbf{X} = \mathbf{x}]$$

we turn to logistic regression. The model is written

$$\log\left(\frac{p(\mathbf{x})}{1 - p(\mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Rearranging, we see the probabilities can be written as

$$p(\mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)}} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)$$

Notice, we use the sigmoid function as shorthand notation, which appears often in deep learning literature. It takes any real input, and outputs a number between 0 and 1. How useful!

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The model is fit by numerically maximizing the likelihood, which we will let `R` take care of.

We start with a single predictor example, again using `balance` as our single predictor.

```
model_glm = glm(default ~ balance, data = default_train, family = "binomial")
```

Fitting this model looks very similar to fitting a simple linear regression. Instead of `lm()` we use `glm()`. The only other difference is the use of `family = "binomial"` which indicates that we have a two-class categorical response. Using `glm()` with `family = "gaussian"` would perform the usual linear regression.

First, we can obtain the fitted coefficients the same way we did with linear regression.

```
coef(model_glm)
```

```
##   (Intercept)       balance
## -10.452182876   0.005367655
```

The next thing we should understand is how the `predict()` function works with `glm()`. So, let's look a some predictions.

```
head(predict(model_glm))
```

```
##       9149       9370       2861       8302       6415       5189
## -6.9616496 -0.7089539 -4.8936916 -9.4123620 -9.0416096 -7.3600645
```

By default, `predict.glm()` uses `type = "link"`.

```
head(predict(model_glm, type = "link"))
```

```
##       9149       9370       2861       8302       6415       5189
## -6.9616496 -0.7089539 -4.8936916 -9.4123620 -9.0416096 -7.3600645
```

That is, R is returning

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p$$

for each observation.

Importantly, these are **not** predicted probabilities. To obtain the predicted probabilities

$$\hat{p}(\mathbf{x}) = \hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}]$$

we need to use `type = "response"`

```
head(predict(model_glm, type = "response"))
```

```
##         9149         9370         2861         8302         6415
## 9.466353e-04 3.298300e-01 7.437969e-03 8.170105e-05 1.183661e-04
##         5189
## 6.357530e-04
```

Note that these are probabilities, **not** classifications. To obtain classifications, we will need to compare to the correct cutoff value with an `ifelse()` statement.

```
model_glm_pred = ifelse(predict(model_glm, type = "link") > 0, "Yes", "No")
# model_glm_pred = ifelse(predict(model_glm, type = "response") > 0.5, "Yes", "No")
```

The line that is run is performing

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & \hat{f}(\mathbf{x}) > 0 \\ 0 & \hat{f}(\mathbf{x}) \leq 0 \end{cases}$$

where

$$\hat{f}(\mathbf{x}) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p.$$

The commented line, which would give the same results, is performing

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & \hat{p}(\mathbf{x}) > 0.5 \\ 0 & \hat{p}(\mathbf{x}) \leq 0.5 \end{cases}$$

where

$$\hat{p}(\mathbf{x}) = \hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}].$$

Once we have classifications, we can calculate metrics such as accuracy.

```
mean(model_glm_pred == default_train$default) # train accuracy
```

```
## [1] 0.9722
```

As we saw previously, the `table()` and `confusionMatrix()` functions can be used to quickly obtain many more metrics.

```
train_tab = table(predicted = model_glm_pred, actual = default_train$default)
library(caret)
train_con_mat = confusionMatrix(train_tab, positive = "Yes")
c(train_con_mat$overall["Accuracy"],
  train_con_mat$byClass["Sensitivity"],
  train_con_mat$byClass["Specificity"])
```

```
##    Accuracy Sensitivity Specificity
##   0.9722000   0.2738095   0.9964818
```

As we did with regression, we could also write a custom function for accuracy.

```
get_accuracy = function(mod, data, res = "y", pos = 1, neg = 0, cut = 0.5) {
  probs = predict(mod, newdata = data, type = "response")
  preds = ifelse(probs > cut, pos, neg)
  mean(data[, res] == preds)
}
```

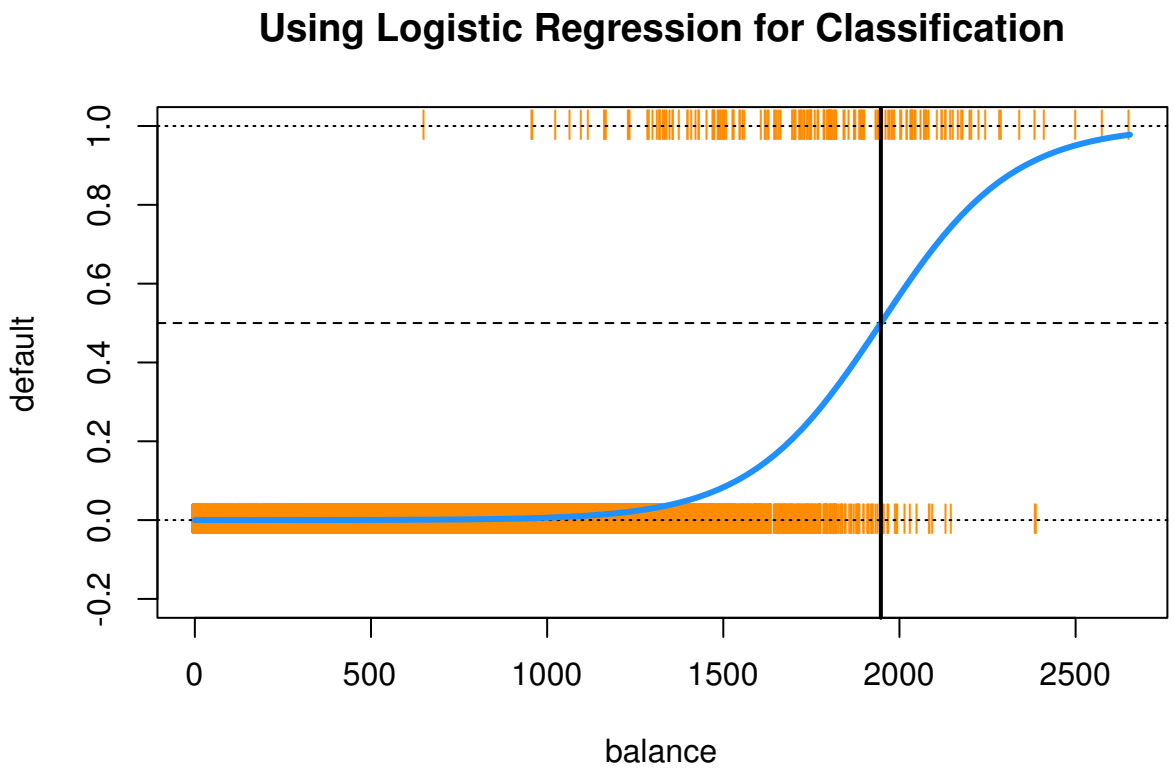
This function will be useful later when calculating train and test accuracies for several models at the same time.

```
get_accuracy(model_glm, data = default_train,
             res = "default", pos = "Yes", neg = "No", cut = 0.5)
```

```
## [1] 0.9722
```

To see how much better logistic regression is for this task, we create the same plot we used for linear regression.

```
plot(default ~ balance, data = default_train_lm,
     col = "darkorange", pch = "|", ylim = c(-0.2, 1),
     main = "Using Logistic Regression for Classification")
abline(h = 0, lty = 3)
abline(h = 1, lty = 3)
abline(h = 0.5, lty = 2)
curve(predict(model_glm, data.frame(balance = x), type = "response"),
      add = TRUE, lwd = 3, col = "dodgerblue")
abline(v = -coef(model_glm)[1] / coef(model_glm)[2], lwd = 2)
```



This plot contains a wealth of information.

- The orange | characters are the data, (x_i, y_i).
- The blue "curve" is the predicted probabilities given by the fitted logistic regression. That is,

$$\hat{p}(\mathbf{x}) = \hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}]$$

- The solid vertical black line represents the **decision boundary**, the `balance` that obtains a predicted probability of 0.5. In this case `balance` = 1947.252994.

The decision boundary is found by solving for points that satisfy

$$\hat{p}(\mathbf{x}) = \hat{P}[Y = 1 \mid \mathbf{X} = \mathbf{x}] = 0.5$$

This is equivalent to point that satisfy

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 = 0.$$

Thus, for logistic regression with a single predictor, the decision boundary is given by the *point*

$$x_1 = \frac{-\hat{\beta}_0}{\hat{\beta}_1}.$$

The following is not run, but an alternative way to add the logistic curve to the plot.

```
grid = seq(0, max(default_train$balance), by = 0.01)

sigmoid = function(x) {
  1 / (1 + exp(-x))
```

```
}

lines(grid, sigmoid(coef(model_glm)[1] + coef(model_glm)[2] * grid), lwd = 3)
```

Using the usual formula syntax, it is easy to add complexity to logistic regressions.

```
model_1 = glm(default ~ 1, data = default_train, family = "binomial")
model_2 = glm(default ~ ., data = default_train, family = "binomial")
model_3 = glm(default ~ . ^ 2 + I(balance ^ 2),
              data = default_train, family = "binomial")
```

Note that, using polynomial transformations of predictors will allow a linear model to have non-linear decision boundaries.

```
model_list = list(model_1, model_2, model_3)

train_error = 1 - sapply(model_list, get_accuracy, data = default_train,
                         res = "default", pos = "Yes", neg = "No", cut = 0.5)
test_error = 1 - sapply(model_list, get_accuracy, data = default_test,
                        res = "default", pos = "Yes", neg = "No", cut = 0.5)
```

Here we see the misclassification error rates for each model. The train decreases, and the test decreases, until it starts to increases. Everything we learned about the bias-variance tradeoff for regression also applies here.

```
diff(train_error)
```

```
## [1] -0.0058 -0.0002
```

```
diff(test_error)
```

```
## [1] -0.0068  0.0004
```

We call `model_2` the **additive** logistic model, which we will use quite often.

8.4 ROC Curves

Let's return to our simple model with only balance as a predictor.

```
model_glm = glm(default ~ balance, data = default_train, family = "binomial")
```

We write a function which allows use to make predictions based on different probability cutoffs.

```
get_pred = function(mod, data, res = "y", pos = 1, neg = 0, cut = 0.5) {
  probs = predict(mod, newdata = data, type = "response")
  ifelse(probs > cut, pos, neg)
}
```

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & \hat{f}(\mathbf{x}) > c \\ 0 & \hat{f}(\mathbf{x}) \leq c \end{cases}$$

Let's use this to obtain predictions using a low, medium, and high cutoff. (0.1, 0.5, and 0.9)

```
test_pred_10 = get_pred(model_glm, data = default_test, res = "default", pos = "Yes", neg = "No", cut =
test_pred_50 = get_pred(model_glm, data = default_test, res = "default", pos = "Yes", neg = "No", cut =
test_pred_90 = get_pred(model_glm, data = default_test, res = "default", pos = "Yes", neg = "No", cut =
```

Now we evaluate accuracy, sensitivity, and specificity for these classifiers.

```
test_tab_10 = table(predicted = test_pred_10, actual = default_test$default)
test_tab_50 = table(predicted = test_pred_50, actual = default_test$default)
test_tab_90 = table(predicted = test_pred_90, actual = default_test$default)

test_con_mat_10 = confusionMatrix(test_tab_10, positive = "Yes")
test_con_mat_50 = confusionMatrix(test_tab_50, positive = "Yes")
test_con_mat_90 = confusionMatrix(test_tab_90, positive = "Yes")
```

```
metrics = rbind(

  c(test_con_mat_10$overall["Accuracy"],
    test_con_mat_10$byClass["Sensitivity"],
    test_con_mat_10$byClass["Specificity"]),

  c(test_con_mat_50$overall["Accuracy"],
    test_con_mat_50$byClass["Sensitivity"],
    test_con_mat_50$byClass["Specificity"]),

  c(test_con_mat_90$overall["Accuracy"],
    test_con_mat_90$byClass["Sensitivity"],
    test_con_mat_90$byClass["Specificity"])

)

rownames(metrics) = c("c = 0.10", "c = 0.50", "c = 0.90")
metrics
```

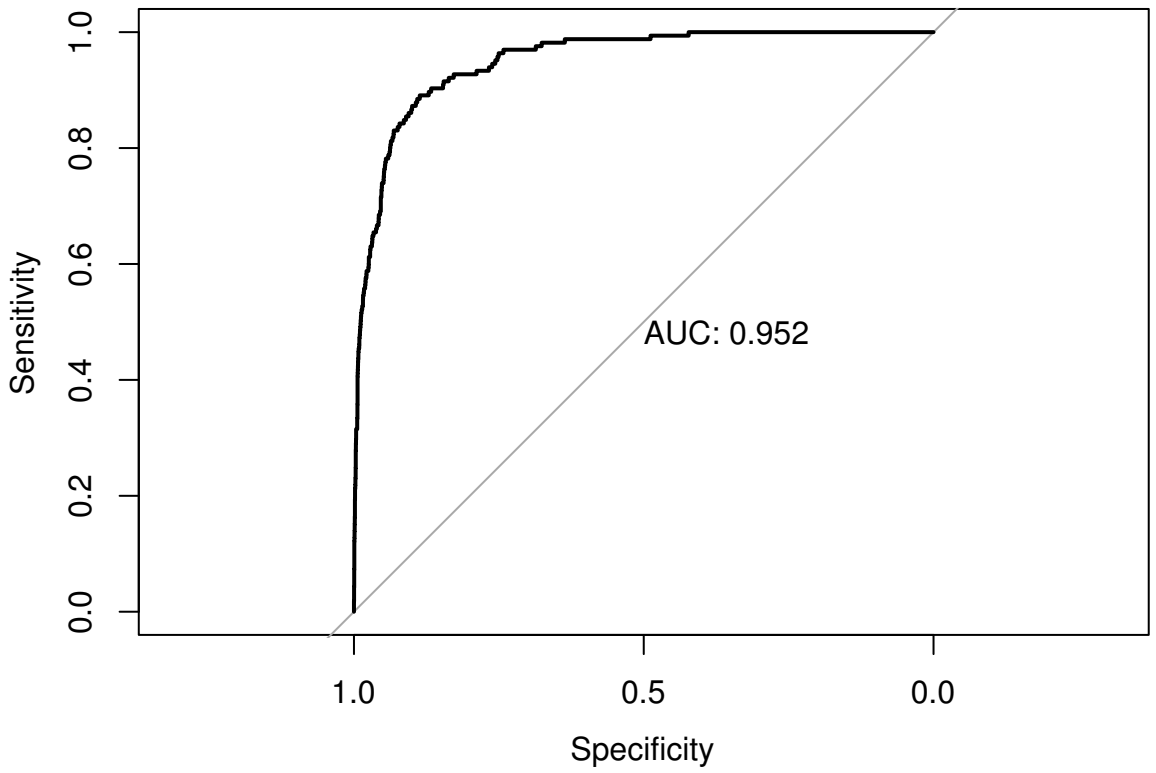
```
##          Accuracy Sensitivity Specificity
## c = 0.10   0.9404  0.77575758   0.9460186
## c = 0.50   0.9738  0.31515152   0.9962771
## c = 0.90   0.9674  0.01818182   0.9997932
```

We see then sensitivity decreases as the cutoff is increased. Conversely, specificity increases as the cutoff increases. This is useful if we are more interested in a particular error, instead of giving them equal weight.

Note that usually the best accuracy will be seen near $c = 0.50$.

Instead of manually checking cutoffs, we can create an ROC curve (receiver operating characteristic curve) which will sweep through all possible cutoffs, and plot the sensitivity and specificity.

```
library(pROC)
test_prob = predict(model_glm, newdata = default_test, type = "response")
test_roc = roc(default_test$default ~ test_prob, plot = TRUE, print.auc = TRUE)
```




```
as.numeric(test_roc$auc)
```

```
## [1] 0.9515076
```

A good model will have a high AUC, that is as often as possible a high sensitivity and specificity.

8.5 Multinomial Logistic Regression

What if the response contains more than two categories? For that we need multinomial logistic regression.

$$P[Y = k \mid \mathbf{X} = \mathbf{x}] = \frac{e^{\beta_{0k} + \beta_{1k} x_1 + \cdots + + \beta_{pk} x_p}}{\sum_{j=1}^{K} e^{\beta_{0j} + \beta_{1j} x_1 + \cdots + + \beta_{pj} x_p}}$$

We will omit the details, as ISL has as well. If you are interested, the Wikipedia page provides a rather thorough coverage. Also note that the above is an example of the softmax function.

As an example of a dataset with a three category response, we use the `iris` dataset, which is so famous, it has its own Wikipedia entry. It is also a default dataset in `R`, so no need to load it.

Before proceeding, we test-train split this data.

```
set.seed(430)
iris_obs = nrow(iris)
iris_index = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_train = iris[iris_index, ]
iris_test = iris[-iris_index, ]
```

To perform multinomial logistic regression, we use the `multinom` function from the `nnet` package. Training using `multinom()` is done using similar syntax to `lm()` and `glm()`. We add the `trace = FALSE` argument to suppress information about updates to the optimization routine as the model is trained.

```
library(nnet)
model_multi = multinom(Species ~ ., data = iris_train, trace = FALSE)
summary(model_multi)$coefficients
```

```
##            (Intercept) Sepal.Length Sepal.Width Petal.Length Petal.Width
## versicolor    26.81602    -6.983313   -16.24574     20.35750    3.218787
## virginica    -34.24228    -8.398869   -17.03985     31.94659   11.594518
```

Notice we are only given coefficients for two of the three class, much like only needing coefficients for one class in logistic regression.

A difference between `glm()` and `multinom()` is how the `predict()` function operates.

```
head(predict(model_multi, newdata = iris_train))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(model_multi, newdata = iris, type = "prob"))
```

```
##   setosa   versicolor    virginica
## 1      1 1.386333e-16 1.137629e-39
## 2      1 1.888634e-12 3.059666e-35
## 3      1 3.868198e-14 2.226923e-37
## 4      1 2.315067e-11 1.687874e-33
## 5      1 5.490420e-17 4.794326e-40
## 6      1 2.196721e-17 1.482366e-38
```

Notice that by default, classifications are returned. When obtaining probabilities, we are given the predicted probability for **each** class.

Interestingly, you've just fit a neural network, and you didn't even know it! (Hence the `nnet` package.) Later we will discuss the connections between logistic regression, multinomial logistic regression, and simple neural networks.

Chapter 9

Generative Models

In this chapter, we continue our discussion of classification methods. We introduce three new methods, each a **generative** method. This in comparison to logistic regression, which is a **discriminative** method.

Generative methods model the joint probability, $p(x, y)$, often by assuming some distribution for the conditional distribution of X given Y, $f(x \mid y)$. Bayes theorem is then applied to classify according to $p(y \mid x)$. Discriminative methods directly model this conditional, $p(y \mid x)$. A detailed discussion and analysis can be found in Ng and Jordan, 2002.

Each of the methods in this chapter will use Bayes theorem to build a classifier.

$$p_k(x) = P[Y = k \mid \mathbf{X} = \mathbf{x}] = \frac{\pi_k \cdot f_k(\mathbf{x})}{\sum_{i=1}^{K} \pi_k \cdot f_k(\mathbf{x})}$$

We call $p_k(x)$ the **posterior** probability, which we will estimate then use to create classifications. The π_k are called the **prior** probabilities for each class k. That is, $P[y = k]$, unconditioned on X. The $f_k(\mathbf{x})$ are called the **likelihoods**, which are indexed by k to denote that they are conditional on the classes. The denominator is often referred to as a **normalizing constant**.

The methods will differ by placing different modeling assumptions on the likelihoods, $f_k(\mathbf{x})$. For each method, the priors could be learned from data or pre-specified.

For each method, classifications are made to the class with the highest estimated posterior probability, which is equivalent to the class with the largest

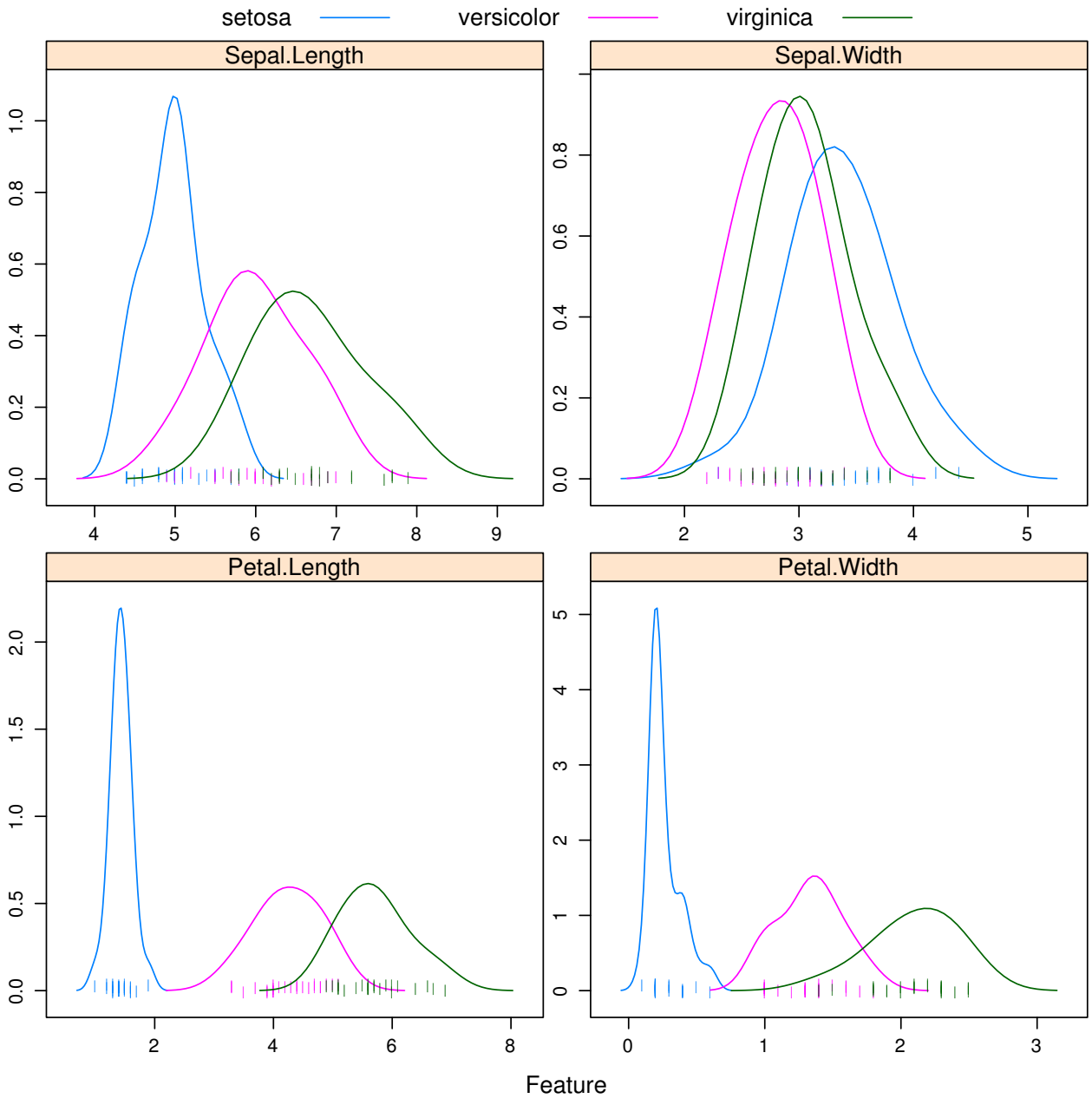
$$\log(\hat{\pi}_k \cdot \hat{f}_k(\mathbf{x})).$$

By substituting the corresponding likelihoods, simplifying, and eliminating unnecessary terms, we could derive the discriminant function for each.

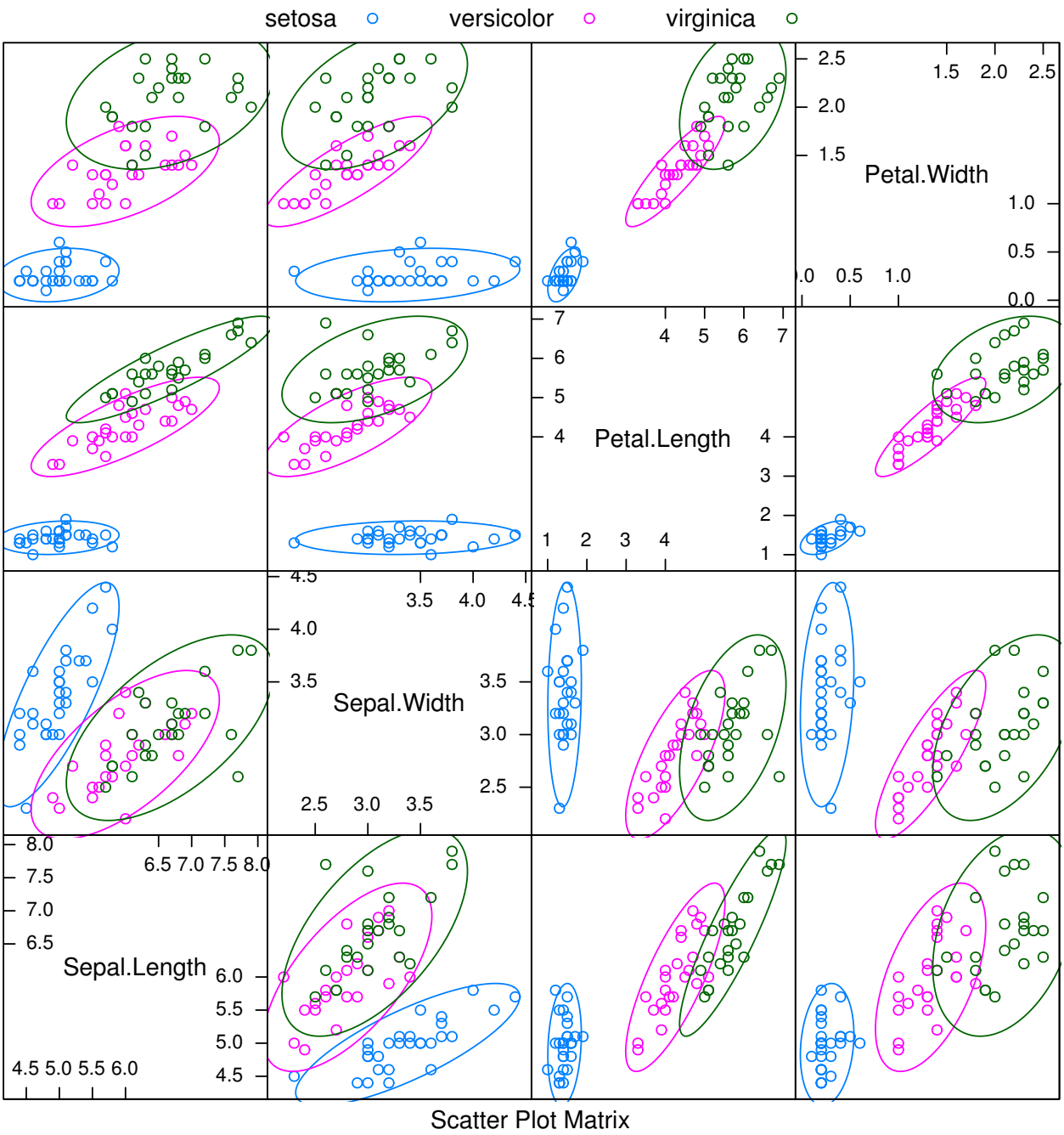
To illustrate these new methods, we return to the iris data, which you may remember has three classes. After a test-train split, we create a number of plots to refresh our memory.

```
set.seed(430)
iris_obs = nrow(iris)
iris_index = sample(iris_obs, size = trunc(0.50 * iris_obs))
# iris_index = sample(iris_obs, size = trunc(0.10 * iris_obs))
iris_train = iris[iris_index, ]
iris_test = iris[-iris_index, ]
```

```
caret::featurePlot(x = iris_train[, c("Sepal.Length", "Sepal.Width",
                                      "Petal.Length", "Petal.Width")],
            y = iris_train$Species,
            plot = "density",
            scales = list(x = list(relation = "free"),
                          y = list(relation = "free")),
            adjust = 1.5,
            pch = "|",
            layout = c(2, 2),
            auto.key = list(columns = 3))
```



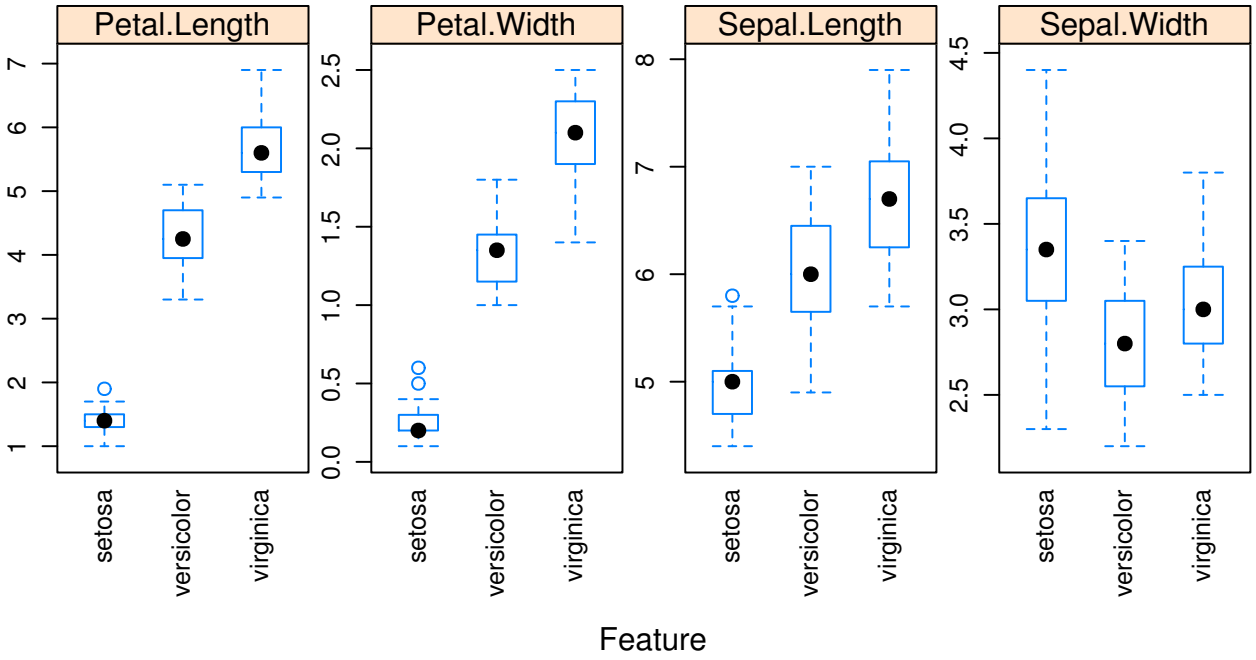
```
caret::featurePlot(x = iris_train[, c("Sepal.Length", "Sepal.Width",
                                      "Petal.Length", "Petal.Width")],
            y = iris_train$Species,
            plot = "ellipse",
            auto.key = list(columns = 3))
```



Scatter Plot Matrix

```
caret::featurePlot(x = iris_train[, c("Sepal.Length", "Sepal.Width",
                                      "Petal.Length", "Petal.Width")],
            y = iris_train$Species,
                  plot = "box",
                  scales = list(y = list(relation="free"),
```

```
                                x = list(rot = 90)),
                layout = c(4, 1))
```



Especially based on the pairs plot, we see that it should not be too difficult to find a good classifier.

Notice that we use `caret::featurePlot` to access the `featurePlot()` function without loading the entire `caret` package.

9.1 Linear Discriminant Analysis

LDA assumes that the predictors are multivariate normal conditioned on the classes.

$$\mathbf{X} \mid Y = k \sim N(\mu_k, \Sigma)$$

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \mu_k)'\Sigma^{-1}(\mathbf{x} - \mu_k)\right]$$

Notice that Σ does **not** depend on k, that is, we are assuming the same Σ for each class. We then use information from all the classes to estimate Σ.

To fit an LDA model, we use the `lda()` function from the `MASS` package.

```
library(MASS)
iris_lda = lda(Species ~ ., data = iris_train)
iris_lda
```

```
## Call:
## lda(Species ~ ., data = iris_train)
##
## Prior probabilities of groups:
```

```
##     setosa versicolor  virginica
##  0.3733333  0.3200000  0.3066667
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.978571    3.378571     1.432143   0.2607143
## versicolor     5.995833    2.808333     4.254167   1.3333333
## virginica      6.669565    3.065217     5.717391   2.0956522
##
## Coefficients of linear discriminants:
##                     LD1        LD2
## Sepal.Length  0.7100013 -0.8446128
## Sepal.Width   1.2435532  2.4773120
## Petal.Length -2.3419418 -0.4065865
## Petal.Width  -1.8502355  2.3234441
##
## Proportion of trace:
##    LD1    LD2
## 0.9908 0.0092
```

Here we see the estimated $\hat{\pi}_k$ and $\hat{\mu}_k$ for each class.

```
is.list(predict(iris_lda, iris_train))
```

```
## [1] TRUE
```

```
names(predict(iris_lda, iris_train))
```

```
## [1] "class"     "posterior" "x"
```

```
head(predict(iris_lda, iris_train)$class, n = 10)
```

```
##  [1] setosa     virginica  setosa     setosa     virginica  setosa
##  [7] virginica  setosa     versicolor setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_lda, iris_train)$posterior, n = 10)
```

```
##           setosa   versicolor    virginica
## 23  1.000000e+00 1.517145e-21 1.717663e-41
## 106 2.894733e-43 1.643603e-06 9.999984e-01
## 37  1.000000e+00 2.169066e-20 1.287216e-40
## 40  1.000000e+00 3.979954e-17 8.243133e-36
## 145 1.303566e-37 4.335258e-06 9.999957e-01
## 36  1.000000e+00 1.947567e-18 5.996917e-38
## 119 2.220147e-51 9.587514e-09 1.000000e+00
## 16  1.000000e+00 5.981936e-23 1.344538e-42
## 94  1.599359e-11 9.999999e-01 1.035129e-07
## 27  1.000000e+00 8.154612e-15 4.862249e-32
```

As we should come to expect, the `predict()` function operates in a new way when called on an `lda` object. By default, it returns an entire list. Within that list `class` stores the classifications and `posterior` contains the estimated probability for each class.

```
iris_lda_train_pred = predict(iris_lda, iris_train)$class
iris_lda_test_pred = predict(iris_lda, iris_test)$class
```

We store the predictions made on the train and test sets.

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

```
accuracy(predicted = iris_lda_train_pred, actual = iris_train$Species)
```

```
## [1] 0.96
```

```
accuracy(predicted = iris_lda_test_pred, actual = iris_test$Species)
```

```
## [1] 0.9866667
```

As expected, LDA performs well on both the train and test data.

```
table(predicted = iris_lda_test_pred, actual = iris_test$Species)
```

```
##             actual
## predicted    setosa versicolor virginica
##   setosa         22          0         0
##   versicolor      0         26         1
##   virginica       0          0        26
```

Looking at the test set, we see that we are perfectly predicting both setosa and versicolor. The only error is labeling a virginica as a versicolor.

```
iris_lda_flat = lda(Species ~ ., data = iris_train, prior = c(1, 1, 1) / 3)
iris_lda_flat
```

```
## Call:
## lda(Species ~ ., data = iris_train, prior = c(1, 1, 1)/3)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
##  0.3333333  0.3333333  0.3333333
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.978571    3.378571     1.432143   0.2607143
## versicolor     5.995833    2.808333     4.254167   1.3333333
## virginica      6.669565    3.065217     5.717391   2.0956522
##
```

```
## Coefficients of linear discriminants:
##                     LD1        LD2
## Sepal.Length  0.7136357 -0.8415442
## Sepal.Width   1.2328623  2.4826497
## Petal.Length -2.3401674 -0.4166784
## Petal.Width  -1.8602343  2.3154465
##
## Proportion of trace:
##    LD1    LD2
## 0.9901 0.0099
```

Instead of learning (estimating) the proportion of the three species from the data, we could instead specify them ourselves. Here we choose a uniform distributions over the possible species. We would call this a "flat" prior.

```
iris_lda_flat_train_pred = predict(iris_lda_flat, iris_train)$class
iris_lda_flat_test_pred = predict(iris_lda_flat, iris_test)$class
```

```
accuracy(predicted = iris_lda_flat_train_pred, actual = iris_train$Species)
```

```
## [1] 0.96
```

```
accuracy(predicted = iris_lda_flat_test_pred, actual = iris_test$Species)
```

```
## [1] 1
```

This actually gives a better test accuracy!

9.2 Quadratic Discriminant Analysis

QDA also assumes that the predictors are multivariate normal conditioned on the classes.

$$\mathbf{X} \mid Y = k \sim N(\mu_k, \Sigma_k)$$

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_k)'\Sigma_k^{-1}(\mathbf{x}-\mu_k)\right]$$

Notice that now Σ_k **does** depend on k, that is, we are allowing a different Σ_k for each class. We only use information from class k to estimate Σ_k.

```
iris_qda = qda(Species ~ ., data = iris_train)
iris_qda
```

```
## Call:
## qda(Species ~ ., data = iris_train)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
```

```
##  0.3733333  0.3200000  0.3066667
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.978571    3.378571     1.432143   0.2607143
## versicolor     5.995833    2.808333     4.254167   1.3333333
## virginica      6.669565    3.065217     5.717391   2.0956522
```

Here the output is similar to LDA, again giving the estimated $\hat{\pi}_k$ and $\hat{\mu}_k$ for each class. Like `lda()`, the `qda()` function is found in the `MASS` package.

Consider trying to fit QDA again, but this time with a smaller training set. (Use the commented line above to obtain a smaller test set.) This will cause an error because there are not enough observations within each class to estimate the large number of parameters in the Σ_k matrices. This is less of a problem with LDA, since all observations, no matter the class, are being use to estimate the shared Σ matrix.

```
iris_qda_train_pred = predict(iris_qda, iris_train)$class
iris_qda_test_pred = predict(iris_qda, iris_test)$class
```

The `predict()` function operates the same as the `predict()` function for LDA.

```
accuracy(predicted = iris_qda_train_pred, actual = iris_train$Species)
```

```
## [1] 0.9866667
```

```
accuracy(predicted = iris_qda_test_pred, actual = iris_test$Species)
```

```
## [1] 0.96
```

```
table(predicted = iris_qda_test_pred, actual = iris_test$Species)
```

```
##             actual
## predicted    setosa versicolor virginica
##   setosa         22          0         0
##   versicolor      0         23         0
##   virginica       0          3        27
```

Here we find that QDA is not performing as well as LDA. It is misclassifying versicolors. Since QDA is a more complex model than LDA (many more parameters) we would say that QDA is overfitting here.

Also note that, QDA creates quadratic decision boundaries, while LDA creates linear decision boundaries. We could also add quadratic terms to LDA to allow it to create quadratic decision boundaries.

9.3 Naive Bayes

Naive Bayes comes in many forms. With only numeric predictors, it often assumes a multivariate normal conditioned on the classes, but a very specific multivariate normal.

$$\mathbf{X} \mid Y = k \sim N(\mu_k, \Sigma_k)$$

Naive Bayes assumes that the predictors $X_1, X_2, \ldots, X_p$ are independent. This is the "naive" part of naive Bayes. The Bayes part is nothing new. Since $X_1, X_2, \ldots, X_p$ are assumed independent, each Σ_k is diagonal, that is, we assume no correlation between predictors. Independence implies zero correlation.

This will allow us to write the (joint) likelihood as a product of univariate distributions. In this case, the product of univariate normal distributions instead of a (joint) multivariate distribution.

$$f_k(\mathbf{x}) = \prod_{j=1}^{j=p} f_{kj}(x_j)$$

Here, $f_{kj}(x_j)$ is the density for the j-th predictor conditioned on the k-th class. Notice that there is a σ_{kj} for each predictor for each class.

$$f_{kj}(x_j) = \frac{1}{\sigma_{kj}\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x_j - \mu_{kj}}{\sigma_{kj}}\right)^2\right]$$

When $p = 1$, this version of naive Bayes is equivalent to QDA.

```
library(e1071)
iris_nb = naiveBayes(Species ~ ., data = iris_train)
iris_nb
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##     setosa versicolor  virginica
##  0.3733333  0.3200000  0.3066667
##
## Conditional probabilities:
##             Sepal.Length
## Y                [,1]      [,2]
##   setosa     4.978571 0.3774742
##   versicolor 5.995833 0.5812125
##   virginica  6.669565 0.6392003
##
##             Sepal.Width
## Y                [,1]      [,2]
##   setosa     3.378571 0.4349177
##   versicolor 2.808333 0.3269313
##   virginica  3.065217 0.3600615
##
##             Petal.Length
## Y                [,1]      [,2]
##   setosa     1.432143 0.1743848
##   versicolor 4.254167 0.5166608
##   virginica  5.717391 0.5540366
##
```

```
##             Petal.Width
## Y                 [,1]      [,2]
##   setosa     0.2607143 0.1133310
##   versicolor 1.3333333 0.2334368
##   virginica  2.0956522 0.3022315
```

Many packages implement naive Bayes. Here we choose to use `naiveBayes()` from the package `e1071`. (The name of this package has an interesting history. Based on the name you wouldn't know it, but the package contains many functions related to machine learning.)

The `Conditional probabilities:` portion of the output gives the mean and standard deviation of the normal distribution for each predictor in each class. Notice how these mean estimates match those for LDA and QDA above.

Note that `naiveBayes()` will work without a factor response, but functions much better with one. (Especially when making predictions.) If you are using a `0` and `1` response, you might consider coercing to a factor first.

```
head(predict(iris_nb, iris_train))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_nb, iris_train, type = "class"))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_nb, iris_train, type = "raw"))
```

```
##             setosa   versicolor    virginica
## [1,]  1.000000e+00 3.134201e-16 2.948226e-27
## [2,] 4.400050e-257 5.188308e-08 9.999999e-01
## [3,]  1.000000e+00 2.263278e-14 1.168760e-24
## [4,]  1.000000e+00 4.855740e-14 2.167253e-24
## [5,] 1.897732e-218 6.189883e-08 9.999999e-01
## [6,]  1.000000e+00 8.184097e-15 6.816322e-26
```

Oh look, `predict()` has another new mode of operation. If only there were a way to unify the `predict()` function across all of these methods...

```
iris_nb_train_pred = predict(iris_nb, iris_train)
iris_nb_test_pred = predict(iris_nb, iris_test)
```

```
accuracy(predicted = iris_nb_train_pred, actual = iris_train$Species)
```

```
## [1] 0.9466667
```

```
accuracy(predicted = iris_nb_test_pred, actual = iris_test$Species)
```

```
## [1] 0.9466667
```

```
table(predicted = iris_nb_test_pred, actual = iris_test$Species)
```

```
##             actual
## predicted    setosa versicolor virginica
##   setosa         22          0         0
##   versicolor      0         26         4
##   virginica       0          0        23
```

Like LDA, naive Bayes is having trouble with virginica.

Method	Train Accuracy	Test Accuracy
LDA	0.9600000	0.9866667
LDA, Flat Prior	0.9600000	1.0000000
QDA	0.9866667	0.9600000
Naive Bayes	0.9466667	0.9466667

Summarizing the results, we see that Naive Bayes is the worst of LDA, QDA, and NB for this data. So why should we care about naive Bayes?

The strength of naive Bayes comes from its ability to handle a large number of predictors, p, even with a limited sample size n. Even with the naive independence assumption, naive Bayes works rather well in practice. Also because of this assumption, we can often train naive Bayes where LDA and QDA may be impossible to train because of the large number of parameters relative to the number of observations.

Here naive Bayes doesn't get a chance to show its strength since LDA and QDA already perform well, and the number of predictors is low. The choice between LDA and QDA is mostly down to a consideration about the amount of complexity needed.

9.4 Discrete Inputs

So far, we have assumed that all predictors are numeric. What happens with categorical predictors?

```
iris_train_mod = iris_train

iris_train_mod$Sepal.Width = ifelse(iris_train$Sepal.Width > 3,
                                    ifelse(iris_train$Sepal.Width > 4,
                                           "Large", "Medium"),
                                    "Small")

unique(iris_train_mod$Sepal.Width)
```

```
## [1] "Medium" "Small"  "Large"
```

Here we make a new dataset where `Sepal.Width` is categorical, with levels `Small`, `Medium`, and `Large`. We then try to train classifiers using only the sepal variables.

```
naiveBayes(Species ~ Sepal.Length + Sepal.Width, data = iris_train_mod)
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
```

```
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##     setosa versicolor  virginica
##  0.3733333  0.3200000  0.3066667
##
## Conditional probabilities:
##             Sepal.Length
## Y                [,1]      [,2]
##   setosa     4.978571 0.3774742
##   versicolor 5.995833 0.5812125
##   virginica  6.669565 0.6392003
##
##             Sepal.Width
## Y                 Large     Medium      Small
##   setosa     0.07142857 0.67857143 0.25000000
##   versicolor 0.00000000 0.25000000 0.75000000
##   virginica  0.00000000 0.43478261 0.56521739
```

Naive Bayes makes a somewhat obvious and intelligent choice to model the categorical variable as a multinomial. It then estimates the probability parameters of a multinomial distribution.

```
lda(Species ~ Sepal.Length + Sepal.Width, data = iris_train_mod)
```

```
## Call:
## lda(Species ~ Sepal.Length + Sepal.Width, data = iris_train_mod)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
##  0.3733333  0.3200000  0.3066667
##
## Group means:
##            Sepal.Length Sepal.WidthMedium Sepal.WidthSmall
## setosa         4.978571         0.6785714        0.2500000
## versicolor     5.995833         0.2500000        0.7500000
## virginica      6.669565         0.4347826        0.5652174
##
## Coefficients of linear discriminants:
##                          LD1        LD2
## Sepal.Length      2.051602  0.4768608
## Sepal.WidthMedium 1.728698 -0.4433340
## Sepal.WidthSmall  3.173903 -2.2804034
##
## Proportion of trace:
##    LD1    LD2
## 0.9764 0.0236
```

LDA however creates dummy variables, here with `Large` is the reference level, then continues to model them as normally distributed. Not great, but better then not using a categorical variable.

Chapter 10

k-Nearest Neighbors

In this chapter we introduce our first **non-parametric** method, k-nearest neighbors, which can be used for both classification and regression.

Each method we have seen so far has been parametric. For example, logistic regression had the form

$$\log\left(\frac{p(\mathbf{x})}{1-p(\mathbf{x})}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

In this case, the β_i are the parameters of the model, which we learned (estimated) by training (fitting) the model.

k-nearest neighbors has no such parameters. Instead, it has a **tuning parameter**, k. This is a parameter which determines *how* the model is trained, instead of a parameter that is *learned* through training. Note that tuning parameters are not used exclusively used with non-parametric methods. Later we will see examples of tuning parameters for parametric methods.

10.1 Classification

```
library(ISLR)
library(class)
library(MASS)
```

We first load some necessary libraries. We'll begin discussing classification by returning to the `Default` data from the `ISLR` package. To illustrate regression, we'll also return to the `Boston` data from the `MASS` package. To perform k-nearest neighbors, we will use the `knn()` function from the `class` package.

10.1.1 Default Data

Unlike many of our previous methods, `knn()` requires that all predictors be numeric, so we coerce `student` to be a `0` and `1` variable instead of a factor. (We can leave the response as a factor.)

```
set.seed(42)
Default$student = as.numeric(Default$student) - 1
default_index = sample(nrow(Default), 5000)
default_train = Default[default_index, ]
default_test = Default[-default_index, ]
```

Also unlike previous methods, `knn()` does not utilize the formula syntax, rather, requires the predictors be their own data frame or matrix, and the class labels be a separate factor variable.

```
# training data
X_default_train = default_train[, -1]
y_default_train = default_train[, 1]

# testing data
X_default_test = default_test[, -1]
y_default_test = default_test[, 1]
```

There is very little "training" with k-nearest neighbors. Essentially the only training is to simply remember the inputs. Because of this, we say that k-nearest neighbors is fast at training time. However, at test time, k-nearest neighbors is very slow. For each test case, the method must find the k-nearest neighbors, which is not computationally cheap. (Note that `knn()` uses Euclidean distance.)

```
head(knn(train = X_default_train,
         test = X_default_test,
         cl = y_default_train,
         k = 3),
     n = 25)
```

```
##  [1] No No No No No No No No No No No No No No No No No No No No No No No No No
## [24] No No
## Levels: No Yes
```

Because of the lack of any need for training, the `knn()` function essentially replaces the `predict()` function, and immediately returns classifications. Here, `knn()` used four arguments:

- `train`, the predictors for the train set.
- `test`, the predictors for the test set. `knn()` will output results for these cases.
- `cl`, the true class labels for the train set.
- `k`, the number of neighbors to consider.

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

We'll use our usual `accuracy()` function to asses how well `knn()` works with this data.

```
accuracy(actual = y_default_test,
         predicted = knn(train = X_default_train,
                         test = X_default_test,
                         cl = y_default_train, k = 5))
```

```
## [1] 0.9684
```

Often with `knn()` we need to consider the scale of the predictors variables. If one variable is contains much larger numbers because of the units or range of the variable, it will dominate other variables in the distance measurements. But this doesn't necessarily mean that it should be such an important variable. It is common practice to scale the predictors to have 0 mean and unit variance. Be sure to apply the scaling to both the train and test data.

```
accuracy(actual = y_default_test,
         predicted = knn(train = scale(X_default_train),
                         test = scale(X_default_test),
                         cl = y_default_train, k = 5))
```

```
## [1] 0.9722
```

Here we see the scaling improves the classification accuracy. This may not always be the case, and often, it is normal to attempt classification with and without scaling.

How do we choose k? Try different values and see which works best.

```
set.seed(42)
k_to_try = 1:100
acc_k = rep(x = 0, times = length(k_to_try))

for(i in seq_along(k_to_try)) {
  pred = knn(train = scale(X_default_train),
             test = scale(X_default_test),
             cl = y_default_train,
             k = k_to_try[i])
  acc_k[i] = accuracy(y_default_test, pred)
}
```

The `seq_along()` function can be very useful for looping over a vector that stores non-consecutive numbers. It often removes the need for an additional counter variable. We actually didn't need it in the above `knn()` example, but it is still a good habit. Here we see an example where we would have otherwise needed another variable.

```
ex_seq = seq(from = 1, to = 100, by = 5)
seq_along(ex_seq)
```

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

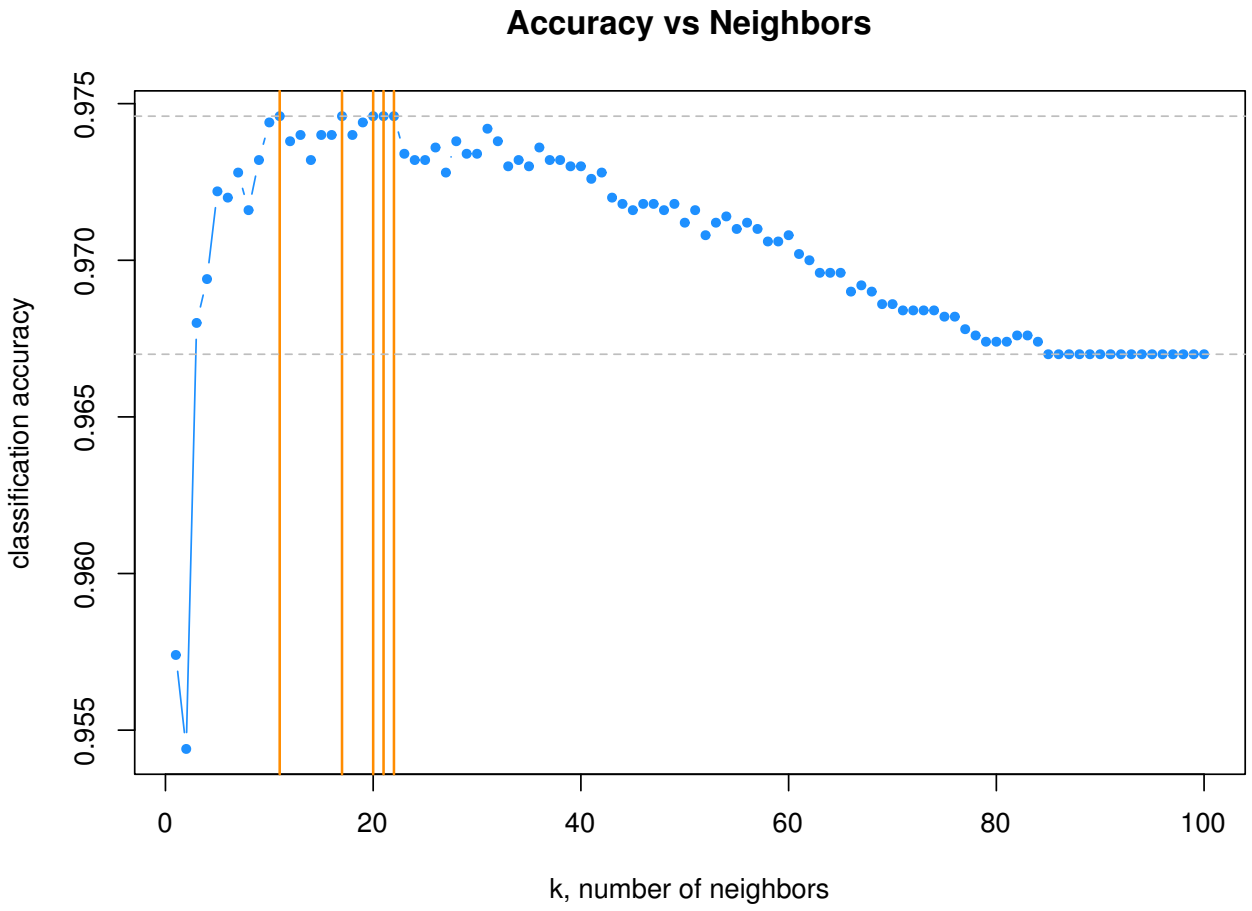
```
ex_storage = rep(x = 0, times = length(ex_seq))
for(i in seq_along(ex_seq)) {
  ex_storage[i] = mean(rnorm(n = 10, mean = ex_seq[i], sd = 1))
}

ex_storage
```

```
##  [1]  0.948629  5.792671 11.090760 15.915397 21.422372 26.106009 30.857772
##  [8] 35.593119 40.958334 46.338667 50.672116 55.733392 60.387860 65.747387
## [15] 71.037306 76.066974 80.956349 85.173316 91.077993 95.882329
```

Naturally, we plot the k-nearest neighbor results.

```
# plot accuracy vs choice of k
plot(acc_k, type = "b", col = "dodgerblue", cex = 1, pch = 20,
     xlab = "k, number of neighbors", ylab = "classification accuracy",
     main = "Accuracy vs Neighbors")
```

```
# add lines indicating k with best accuracy
abline(v = which(acc_k == max(acc_k)), col = "darkorange", lwd = 1.5)
# add line for max accuracy seen
abline(h = max(acc_k), col = "grey", lty = 2)
# add line for prevalence in test set
abline(h = mean(y_default_test == "No"), col = "grey", lty = 2)
```




```
max(acc_k)
```

```
## [1] 0.9746
```

```
max(which(acc_k == max(acc_k)))
```

```
## [1] 22
```

We see that four different values of k are tied for the highest accuracy. Given a choice of these four values of k, we select the largest, as it is the least variable, and has the least chance of overfitting.

Also notice that, as k increases, eventually the accuracy approaches the test prevalence.

```
mean(y_default_test == "No")
```

```
## [1] 0.967
```

10.1.2 Iris Data

Like LDA and QDA, KNN can be used for both binary and multi-class problems. As an example, we return to the iris data.

```
set.seed(430)
iris_obs = nrow(iris)
iris_index = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_train = iris[iris_index, ]
iris_test = iris[-iris_index, ]
```

All the predictors here are numeric, so we proceed to splitting the data into predictors and classes.

```
# training data
X_iris_train = iris_train[, -5]
y_iris_train = iris_train[, 5]

# testing data
X_iris_test = iris_test[, -5]
y_iris_test = iris_test[, 5]
```

Like previous methods, we can obtain predicted probabilities given test predictors. To do so, we add an argument, `prob = TRUE`

```
iris_pred = knn(train = scale(X_iris_train),
                test = scale(X_iris_test),
                cl = y_iris_train,
                k = 10,
                prob = TRUE)
```

```
iris_pred
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     setosa     setosa     setosa
## [19] setosa     setosa     setosa     setosa     versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] versicolor versicolor versicolor versicolor versicolor versicolor
## [37] versicolor versicolor versicolor versicolor versicolor versicolor
## [43] versicolor versicolor versicolor versicolor versicolor versicolor
## [49] virginica  versicolor virginica  virginica  virginica  virginica
## [55] virginica  virginica  virginica  versicolor versicolor virginica
## [61] virginica  virginica  virginica  versicolor virginica  virginica
## [67] virginica  virginica  virginica  versicolor virginica  virginica
## [73] virginica  virginica  versicolor
## attr(,"prob")
##  [1] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
##  [8] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [15] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [22] 1.0000000 0.9000000 1.0000000 0.8000000 1.0000000 0.9000000 0.9000000
## [29] 0.9000000 0.8000000 1.0000000 0.9000000 1.0000000 0.8000000 0.5000000
## [36] 0.8000000 0.9000000 0.8000000 1.0000000 1.0000000 0.7272727 0.9000000
```

```
## [43] 0.8000000 0.9000000 1.0000000 1.0000000 0.9000000 0.9000000 0.9000000
## [50] 0.7000000 0.8000000 0.7272727 0.8000000 0.8000000 0.8000000 0.9000000
## [57] 0.6000000 0.6000000 0.5000000 0.9000000 0.6000000 1.0000000 0.6000000
## [64] 0.5000000 0.7000000 0.9000000 1.0000000 0.9000000 0.6000000 0.7000000
## [71] 0.8000000 0.9000000 0.8000000 0.9000000 0.5000000
## Levels: setosa versicolor virginica
```

Unfortunately, this only returns the predicted probability of the most common class. In the binary case, this would be sufficient, however, for multi-class problems, we cannot recover each of the probabilities of interest.

```
attributes(iris_pred)$prob
```

```
##  [1] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
##  [8] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [15] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
## [22] 1.0000000 0.9000000 1.0000000 0.8000000 1.0000000 0.9000000 0.9000000
## [29] 0.9000000 0.8000000 1.0000000 0.9000000 1.0000000 0.8000000 0.5000000
## [36] 0.8000000 0.9000000 0.8000000 1.0000000 1.0000000 0.7272727 0.9000000
## [43] 0.8000000 0.9000000 1.0000000 1.0000000 0.9000000 0.9000000 0.9000000
## [50] 0.7000000 0.8000000 0.7272727 0.8000000 0.8000000 0.8000000 0.9000000
## [57] 0.6000000 0.6000000 0.5000000 0.9000000 0.6000000 1.0000000 0.6000000
## [64] 0.5000000 0.7000000 0.9000000 1.0000000 0.9000000 0.6000000 0.7000000
## [71] 0.8000000 0.9000000 0.8000000 0.9000000 0.5000000
```

10.2 Regression

We quickly illustrate KNN for regression using the `Boston` data. We'll only use `lstat` as a predictor, and `medv` as the response. We won't test-train split for this example since won't be checking RMSE, but instead plotting fitted models. There is also no need to worry about scaling since there is only one predictor.

```
X_boston = Boston["lstat"]
y_boston = Boston["medv"]
```

We create a "test" set, that is a grid of `lstat` values at which we will predict `medv`.

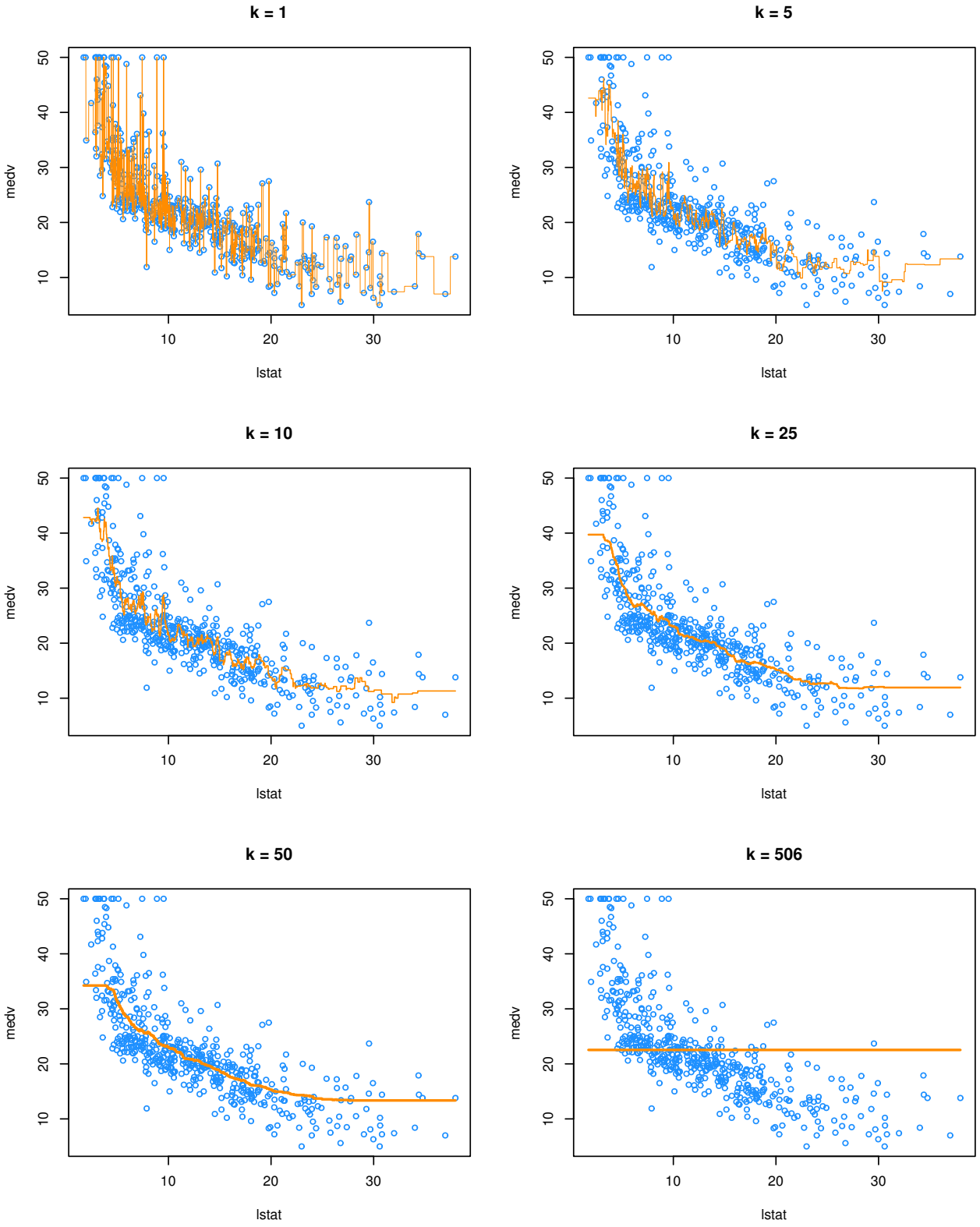
```
lstat_grid = data.frame(lstat = seq(range(X_boston$lstat)[1], range(X_boston$lstat)[2], by = 0.01))
```

Unfortunately, `knn()` from `class` only handles classification. To perform regression, we will need `knn.reg()` from the `FNN` package. Notice that, we do **not** load this package, but instead use `FNN::knn.reg` to access the function. This is useful since `FNN` also contains a function `knn()` and would then mask `knn()` from `class`.

```
pred_001 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 1)
pred_005 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 5)
pred_010 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 10)
pred_050 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 50)
pred_100 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 100)
pred_506 = FNN::knn.reg(train = X_boston, test = lstat_grid, y = y_boston, k = 506)
```

We make predictions for various values of `k`. Note that `506` is the number of observations in this dataset.

```
par(mfrow = c(3, 2))

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 1")
lines(lstat_grid$lstat, pred_001$pred, col = "darkorange", lwd = 0.25)

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 5")
lines(lstat_grid$lstat, pred_005$pred, col = "darkorange", lwd = 0.75)

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 10")
lines(lstat_grid$lstat, pred_010$pred, col = "darkorange", lwd = 1)

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 25")
lines(lstat_grid$lstat, pred_050$pred, col = "darkorange", lwd = 1.5)

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 50")
lines(lstat_grid$lstat, pred_100$pred, col = "darkorange", lwd = 2)

plot(medv ~ lstat, data = Boston, cex = .8, col = "dodgerblue", main = "k = 506")
lines(lstat_grid$lstat, pred_506$pred, col = "darkorange", lwd = 2)
```



We see that `k = 1` is clearly overfitting, as `k = 1` is a very complex, highly variable model. Conversely, `k = 506` is clearly underfitting the data, as `k = 506` is a very simple, low variance model. In fact, here it is predicting a simple average of all the data at each point.

Bibliography