R for Statistical Learning

David Dalpiaz

2017-09-18

Contents

0.1 About This Book . . . 9
0.2 Organization . . . 9
0.3 Who? . . . 9
0.4 Caveat Emptor . . . 10
0.5 Conventions . . . 10
0.6 Acknowledgements . . . 10
0.7 License . . . 11

I Prerequisites **13**

1 Overview **15**

2 Probability Review **17**
2.1 Probability Models . . . 17
2.2 Probability Axioms . . . 17
2.3 Probability Rules . . . 18
2.4 Random Variables . . . 19
2.4.1 Distributions . . . 19
2.4.2 Discrete Random Variables . . . 19
2.4.3 Continuous Random Variables . . . 20
2.4.4 Several Random Variables . . . 21
2.5 Expectations . . . 21
2.6 Likelihood . . . 22
2.7 Videos . . . 22
2.8 References . . . 22

3 R, RStudio, RMarkdown **23**
3.1 Videos . . . 23
3.2 Template . . . 23

4 Modeling Basics in R **25**

4.1 Visualization for Regression . . . 26
4.2 The `lm()` Function . . . 28
4.3 Hypothesis Testing . . . 28
4.4 Prediction . . . 29
4.5 Unusual Observations . . . 30
4.6 Adding Complexity . . . 30
4.6.1 Interactions . . . 31
4.6.2 Polynomials . . . 32
4.6.3 Transformations . . . 33
4.7 `rmarkdown` . . . 33

II Regression **35**

5 Overview **37**

6 Linear Models **39**

6.1 Assesing Model Accuracy . . . 40
6.2 Model Complexity . . . 41
6.3 Test-Train Split . . . 41
6.4 Adding Flexibility to Linear Models . . . 43
6.5 Choosing a Model . . . 44

7 k-Nearest Neighbors **47**

7.1 k-Nearest Neighbors . . . 47
7.2 KNN in R . . . 47

8 Simulating the Bias–Variance Tradeoff **51**

8.1 Bias-Variance Decomposition . . . 51
8.2 Simulation . . . 51
8.3 Bias-Variance Tradeoff . . . 55

III Classification **59**

9 Overview **61**

9.1 Visualization for Classification . . . 62
9.2 A Simple Classifier . . . 66
9.3 Metrics for Classification . . . 67

10 Logistic Regression **71**
10.1 Linear Regression . . . 71
10.2 Bayes Classifier . . . 73
10.3 Logistic Regression with `glm()` . . . 74
10.4 ROC Curves . . . 78
10.5 Multinomial Logistic Regression . . . 80

11 Generative Models **83**
11.1 Linear Discriminant Analysis . . . 86
11.2 Quadratic Discriminant Analysis . . . 89
11.3 Naive Bayes . . . 90
11.4 Discrete Inputs . . . 93
11.5 RMarkdown . . . 94

12 k-Nearest Neighbors **97**
12.1 Classification . . . 97
12.1.1 Default Data . . . 97
12.1.2 Iris Data . . . 101
12.2 Regression . . . 102
12.3 External Links . . . 105
12.4 RMarkdown . . . 105

IV Unsupervised Learning **107**

13 Overview **109**
13.1 Methods . . . 109
13.1.1 Principal Component Analysis . . . 109
13.1.2 k-Means Clustering . . . 109
13.1.3 Hierarchical Clustering . . . 109
13.2 Examples . . . 109
13.2.1 US Arrests . . . 109
13.2.2 Simulated Data . . . 116
13.2.3 Iris Data . . . 125
13.3 External Links . . . 131
13.4 RMarkdown . . . 131

14 Principal Component Analysis **133**

15 k-Means **135**

16 Mixture Models **137**

17 Hierarchical Clustering **139**

V In Practice 141

18 Overview **143**

19 Supervised Learning Overview **145**

Discriminative versus Generative Methods . . . 146

19.1 External Links . . . 146

19.2 RMarkdown . . . 147

20 Resampling **149**

20.1 Test-Train Split . . . 150

20.2 Cross-Validation . . . 151

20.2.1 Method Specific . . . 151

20.2.2 Manual Cross-Validation . . . 155

20.2.3 Test Data . . . 156

20.3 Bootstrap . . . 159

20.4 External Links . . . 159

20.5 RMarkdown . . . 159

21 The `caret` Package **161**

21.1 External Links . . . 169

21.2 RMarkdown . . . 169

22 Subset Selection **171**

22.1 AIC, BIC, and Cp . . . 171

22.1.1 `leaps` Package . . . 171

22.1.2 Best Subset . . . 171

22.1.3 Stepwise Methods . . . 174

22.2 Validated RMSE . . . 175

22.3 External Links . . . 178

22.4 RMarkdown . . . 178

VI The Modern Era 179

23 Overview **181**

24 Regularization **183**

24.1 Ridge Regression . . . 184
24.2 Lasso . . . 189
24.3 `broom` . . . 194
24.4 Simulation Study, p > n . . . 196
24.5 External Links . . . 200
24.6 RMarkdown . . . 200

25 Elastic Net **203**

25.1 Hitters Data . . . 203
25.2 Elastic Net for Regression . . . 204
25.3 Elastic Net for Classification . . . 207
25.4 External Links . . . 208
25.5 RMarkdown . . . 209

26 Trees **211**

26.1 Classification Trees . . . 211
26.2 Regression Trees . . . 219
26.3 `rpart` Package . . . 224
26.4 External Links . . . 228
26.5 RMarkdown . . . 228

27 Ensemble Methods **231**

27.1 Regression . . . 231
27.1.1 Tree Model . . . 231
27.1.2 Linear Model . . . 232
27.1.3 Bagging . . . 233
27.1.4 Random Forest . . . 235
27.1.5 Boosting . . . 236
27.1.6 Results . . . 239
27.2 Classification . . . 239
27.2.1 Tree Model . . . 240
27.2.2 Logistic Regression . . . 240
27.2.3 Bagging . . . 241
27.2.4 Random Forest . . . 241
27.2.5 Boosting . . . 242
27.2.6 Results . . . 242
27.3 Tuning . . . 243

27.3.1 Random Forest and Bagging 243
27.3.2 Boosting 245
27.4 Tree versus Ensemble Boundaries 246
27.5 External Links 249
27.6 RMarkdown 250

28 Artificial Neural Networks **251**

VII Appendix 253

29 Overview **255**

30 Non-Linear Models **257**

31 Regularized Discriminant Analysis **259**

32 Support Vector Machines **261**

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.

Currently the focus of this text is to expand on ISL's introduction to using R for statistical learning. Eventually the text may become more self-contained.

0.2 Organization

The text is organized into seven parts.

1. Prerequisites
2. (Supervised Learning) Regression
3. (Supervised Learning) Classification
4. Unsupervised Learning
5. (Statistical Learning) in Practice
6. (Statistical Learning) in The Modern Era
7. Appendix

Part 1 details the assumed prerequisite knowledge required to read the text. It recaps some of the more important bits of information.

Parts 2, 3, and 4 discuss the theory of statistical learning. Several methods are introduced throughout to highlight different theoretical concepts.

Parts 5 and 6 highlight the use of statistical learning in practice. Part 5 focuses on practical usage of the techniques seen in Parts 2, 3, and 4. Part 6 introduces techniques that are most commonly used in practice today.

0.3 Who?

This book is targeted at advanced undergraduate or first year MS students in Statistics who have no prior statistical learning experience. While both will be discussed in great detail, previous experience with both statistical modeling and R are assumed.

0.4 Caveat Emptor

This "book" is under active development. Much of the text was hastily written during the Spring 2017 run of the course. While together with ISL the coverage is essentially complete, significant updates will occur during Fall 2017.

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.

You will often see "**TODO**" scattered throughout the text. These are mostly notes for internal use, but give the reader some idea of what development is still to come.

0.5 Conventions

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.

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

`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 symbol $\triangleq$ will be used to mean "is defined to be."

We use the value p to mean the number of **p**redictors.

0.6 Acknowledgements

Your name could be here! Suggest an edit! Correct a typo! Pull requests encouraged! If you submit a correction and would like to be listed below, please provide your name as you would like it to appear, as well as a link to a GitHub, LinkedIn, or personal website.

- James Balamuta, Summer 2016 - ???
- Korawat Tanwisuth, Spring 2017
- Yiming Gao, Spring 2017
- Binxiang Ni, Summer 2017
- Ruiqi (Zoe) Li, Summer 2017
- Rachel Banoff, Fall 2017

0.7 License

Part I

Prerequisites

Chapter 1

Overview

TODO: Discussion of necessary knowledge before reading book. Explain what will be recapped along the way.

Chapter 2

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.

2.1 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.

2.2 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.

2.3 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

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

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

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

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

2.4 Random Variables

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

2.4.1 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*.

2.4.2 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 amount 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 **parameters** 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

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

2.4.3 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

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

2.4.4 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.

2.5 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

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

2.6 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.

2.7 Videos

The YouTube channel mathematicalmonk has a great Probability Primer playlist containing lectures on many fundamental probability concepts. Some of the more important concepts are covered in the following videos:

- Conditional Probability
- Independence
- More Independence
- Bayes Rule

2.8 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 3

R, RStudio, RMarkdown

Materials for leaning R, RStudio, and RMarkdown can be found in another text from the same author, Applied Statistics with R.

The chapters up to and including Chapter 6 - R Resources contain an introduction to using R, RStudio, and RMarkdown. This chapter in particular contains a number of videos to get you up to speed on R, RStudio, and RMarkdown, which are also linked below. Also linked is an RMarkdown template which is referenced in the videos.

3.1 Videos

- R and RStudio Playlist
- Data in R Playlist
- RMarkdown Playlist

3.2 Template

- RMarkdown Template

Chapter 4

Modeling Basics in R

TODO: Instead of specifically considering regression, change the focus of this chapter to modeling, with regression as an example.

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 x 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
```

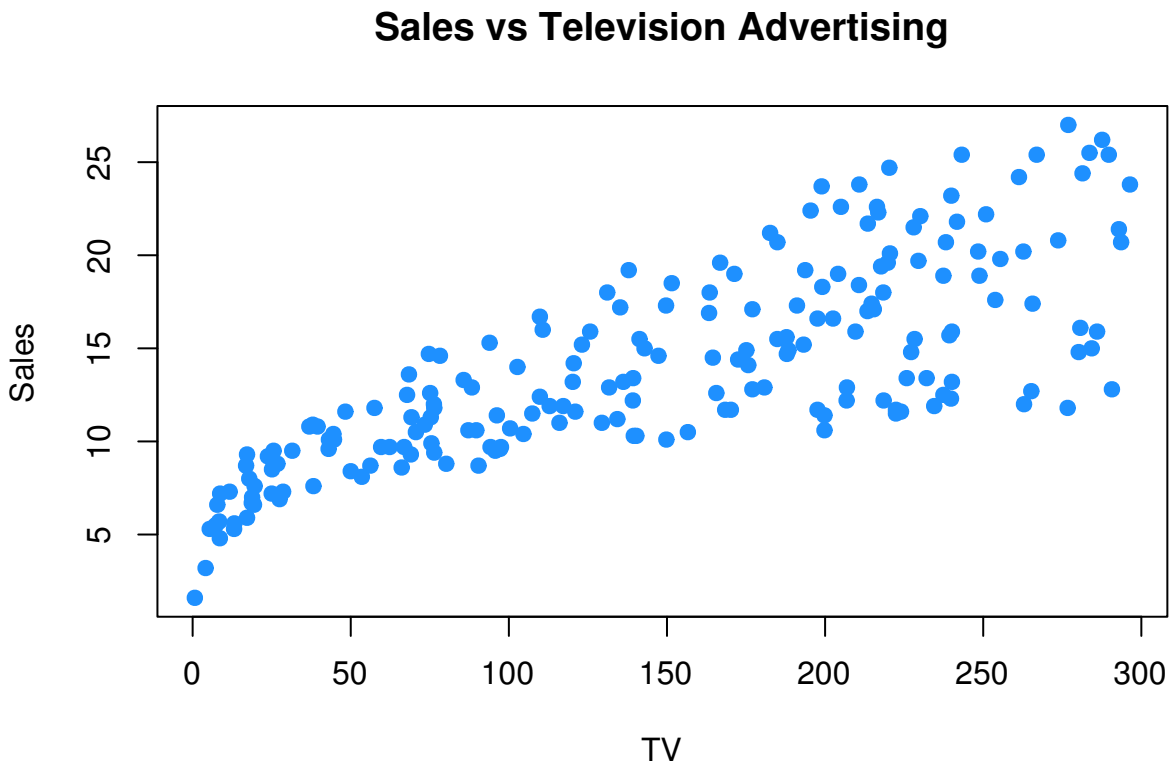
Because the data was read using `read_csv()`, `Advertising` is a tibble. We see that there are a total of 200 observations and 4 variables, each of which is numeric. (Specifically double-precision vectors, but more importantly they are numbers.) For the purpose of this analysis, `Sales` will be the **response 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**.

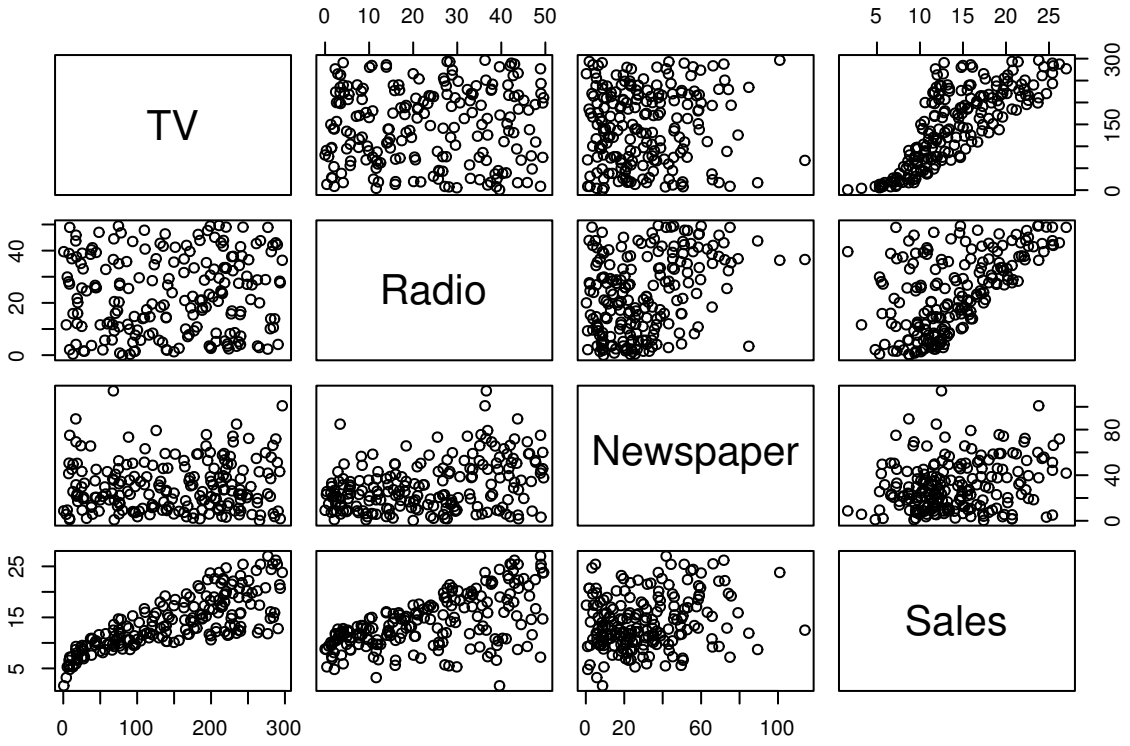
We could do so for any individual predictor.

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



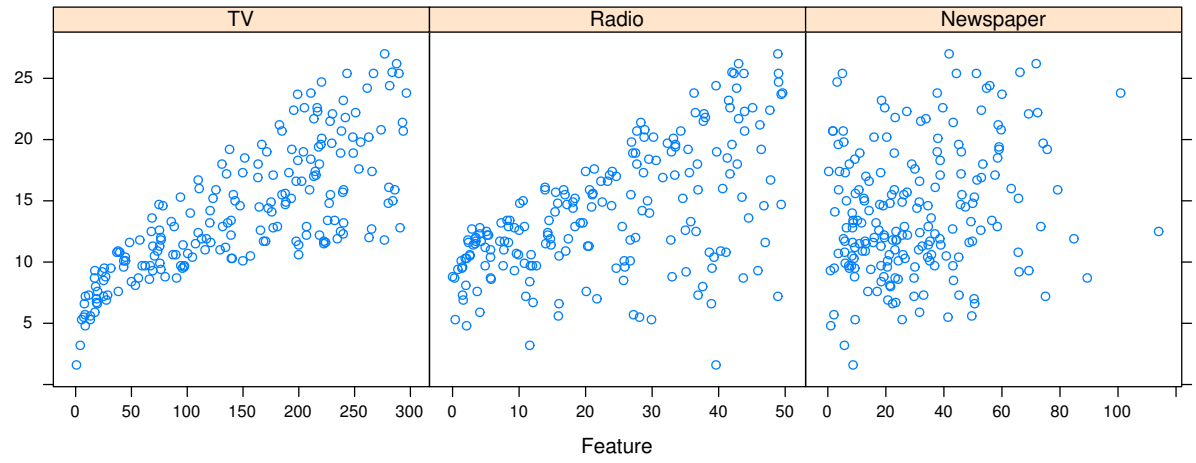
The `pairs()` function is a useful way to quickly visualize a number of scatter plots.

```
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 x 3
##        x1         x2          y
##    <fctr>      <dbl>      <dbl>
##  1      A 0.73666287 -1.4069170
```

```
##  2      A 0.84045952 -1.2949993
##  3      A 0.86646223  0.2304368
##  4      A 0.91392074  0.1183187
##  5      A 0.80408092  2.6896025
##  6      A 0.53699146  2.0695882
##  7      A 0.04861602  1.5339024
##  8      A 0.86022834  0.2199970
##  9      A 0.13351138  0.7552720
## 10      A 0.12566353  1.0818659
## # ... 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 categorical 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
##   1.1157505  -0.8293327  -0.7216672  -0.8796305
```

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

```
## (Intercept)         x1B         x1C          x2      x1B:x2      x1C:x2
##   1.4288455  -1.0827619  -2.1251538  -1.4133215   0.3792214   2.2280549
```

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
```

4.7 rmarkdown

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.4.1. The following packages (and their dependencies) were loaded in this file:

```
## [1] "tibble"  "caret"   "ggplot2" "lattice" "readr"
```

Part II

Regression

Chapter 5

Overview

TODO: Take main concepts from the next chapter and talk about them in general here. The next chapter will illustrate them by example.

Note: This is a placeholder chapter that is new.

- **Supervised Learning**
  - **Regression** (Numeric Response)
    - What do we want? To make *predictions* on *unseen data*. (Predicting on data we already have is easy...) In other words, we want a *model* that **generalizes** well. That is, generalizes to unseen data.
    - How we will do this? By controlling the **complexity** of the model to guard against **overfitting** and **underfitting**.
      - Model Parameters
      - Tuning Parameters
    - Why does manipulating the model complexity accomplish this? Because there is a **bias-variance tradeoff**.
    - How do we know if our model generalizes? By evaluating *metrics* on **test** data. We will only ever fit (train) models on training data. All analyses will begin with a test-train split. For regression tasks, our metric will be **RMSE**.
  - Classification (Categorical Response) The next section.

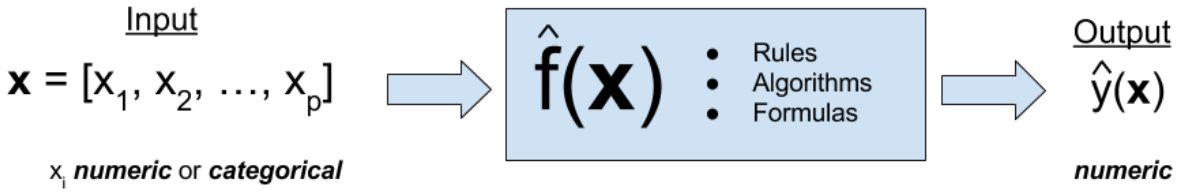


Figure 5.1:

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 (model) we learn (fit, train), 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.

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

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

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

Chapter 6

Linear Models

TODO: Ungeneralize this chapter. Move some specifics to following chapter on use of linear models.

NOTE: This chapter has previously existed. Eventually the concepts will be first introduced in the preceding chapter.

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.

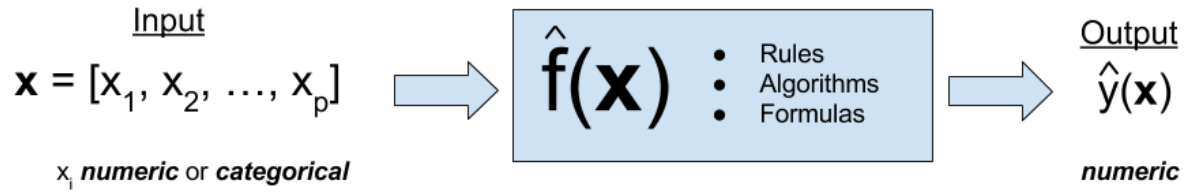


Figure 6.1:

First, note that a linear model is one of many methods used in regression.

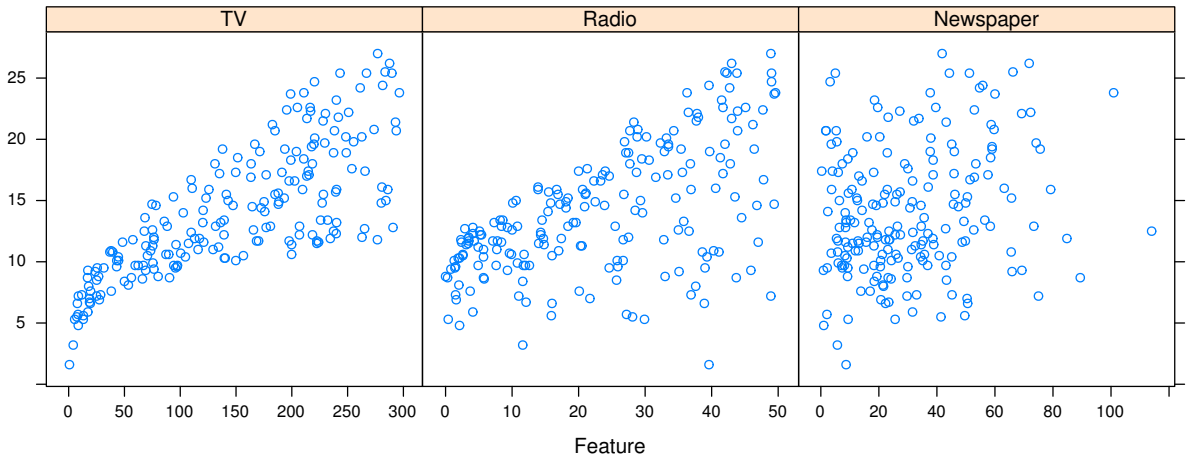
To discuss linear models in the context of prediction, we return to the `Advertising` data from the previous chapter.

```
Advertising
```

```
## # A tibble: 200 x 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)
```



6.1 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.

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

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

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

6.2 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
}
```

6.3 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.

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

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
```

6.4 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
```

6.5 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.

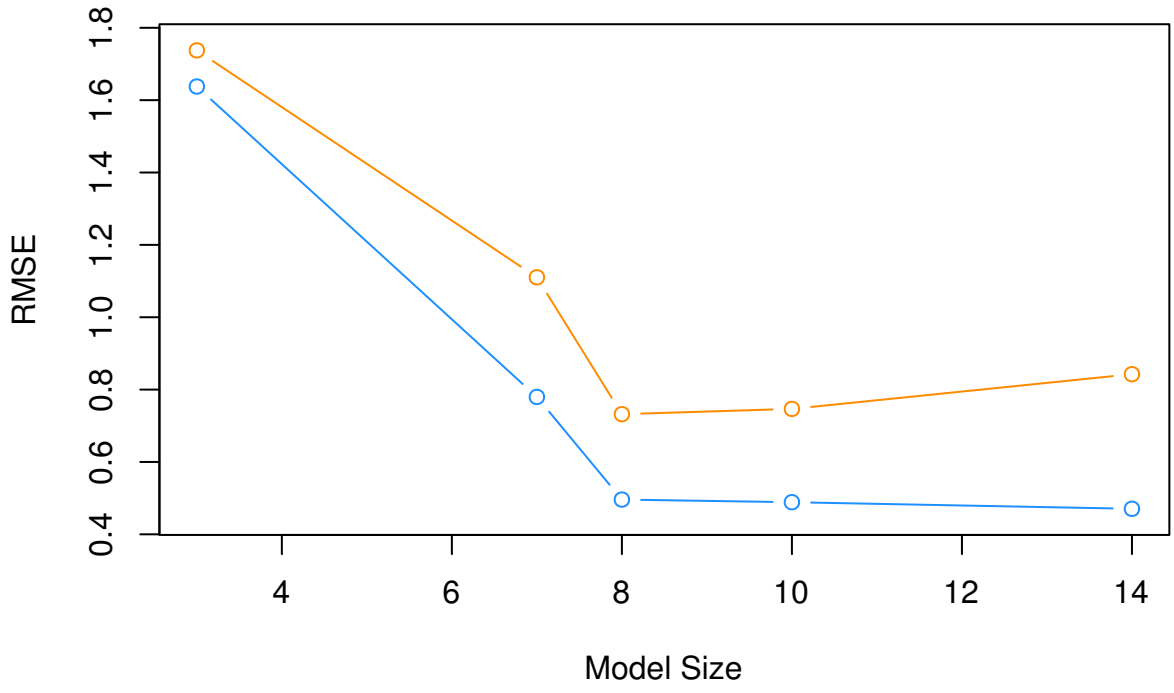
```
model_list = list(fit_1, fit_2, fit_3, fit_4, fit_5)
```

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 7

k-Nearest Neighbors

7.1 k-Nearest Neighbors

$$\hat{f}(x) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x, \mathcal{D})} y_i$$

7.2 KNN in R

```
library(FNN)
library(MASS)
data(Boston)
```

```
set.seed(420)
boston_idx = sample(1:nrow(Boston), size = 250)
trn_boston = Boston[boston_idx, ]
tst_boston  = Boston[-boston_idx, ]
```

```
X_trn_boston = trn_boston["lstat"]
X_tst_boston = tst_boston["lstat"]
y_trn_boston = trn_boston["medv"]
y_tst_boston = tst_boston["medv"]
```

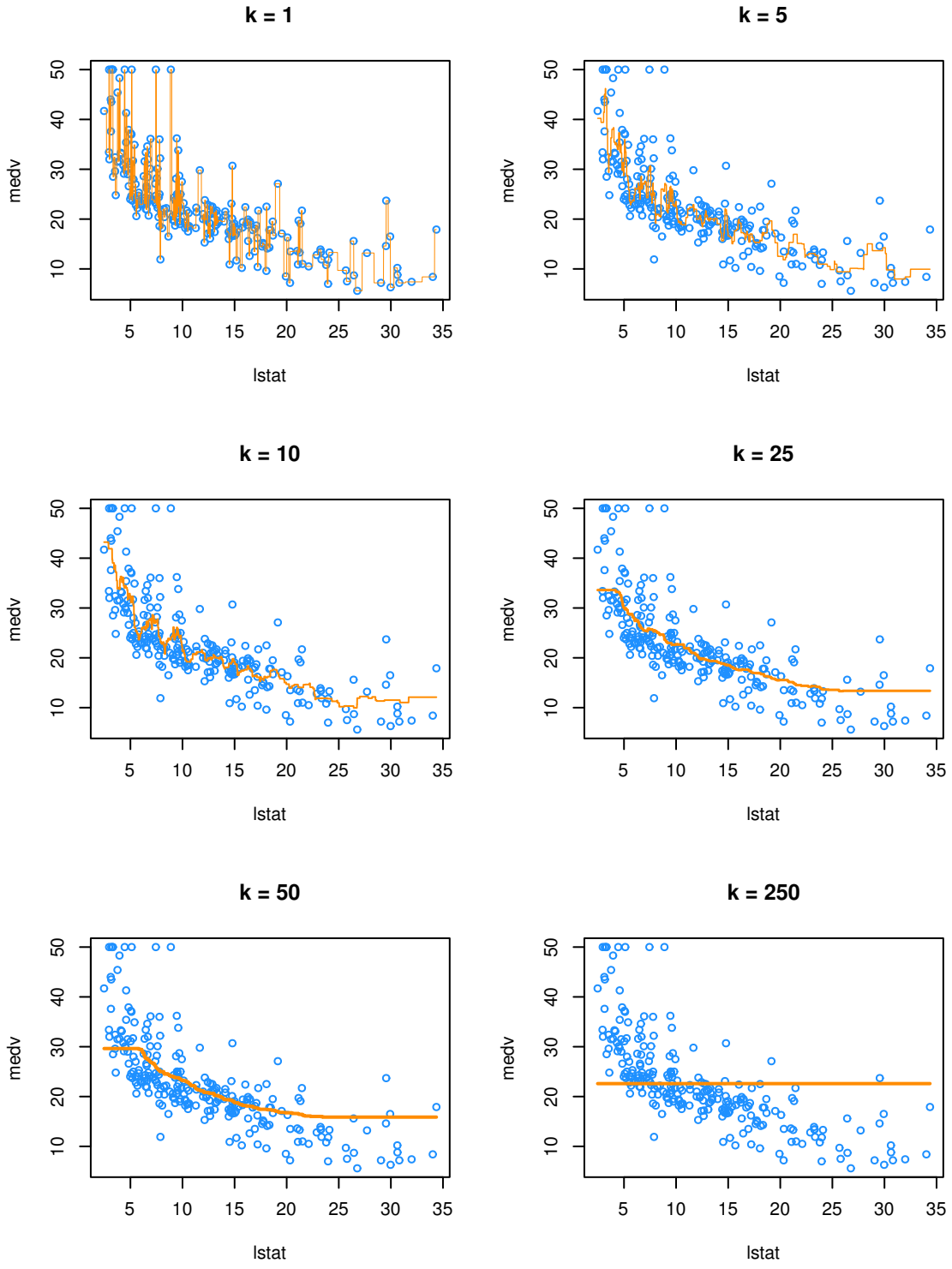
We create an additional "test" set `lstat_grid`, that is a grid of `lstat` values at which we will predict `medv` in order to create graphics.

```
X_trn_boston_min = min(X_trn_boston)
X_trn_boston_max = max(X_trn_boston)
lstat_grid = data.frame(lstat = seq(X_trn_boston_min, X_trn_boston_max,
                                    by = 0.01))
```

To perform KNN for 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. Note that, in the future, we'll need to be care full about loading the `FNN` package as it also contains a function called `knn`. This function also appears in the `class` package which we will likely use later.

```
pred_001 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 1)
pred_005 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 5)
pred_010 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 10)
pred_050 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 50)
pred_100 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 100)
pred_250 = knn.reg(train = X_trn_boston, test = lstat_grid, y = y_trn_boston, k = 250)
```

We make predictions for various values of `k`. Note that `250` is the total number of observations in this training dataset.



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.

Chapter 8

Simulating the Bias–Variance Tradeoff

Consider the general regression setup

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

with

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

8.1 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.

8.2 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.

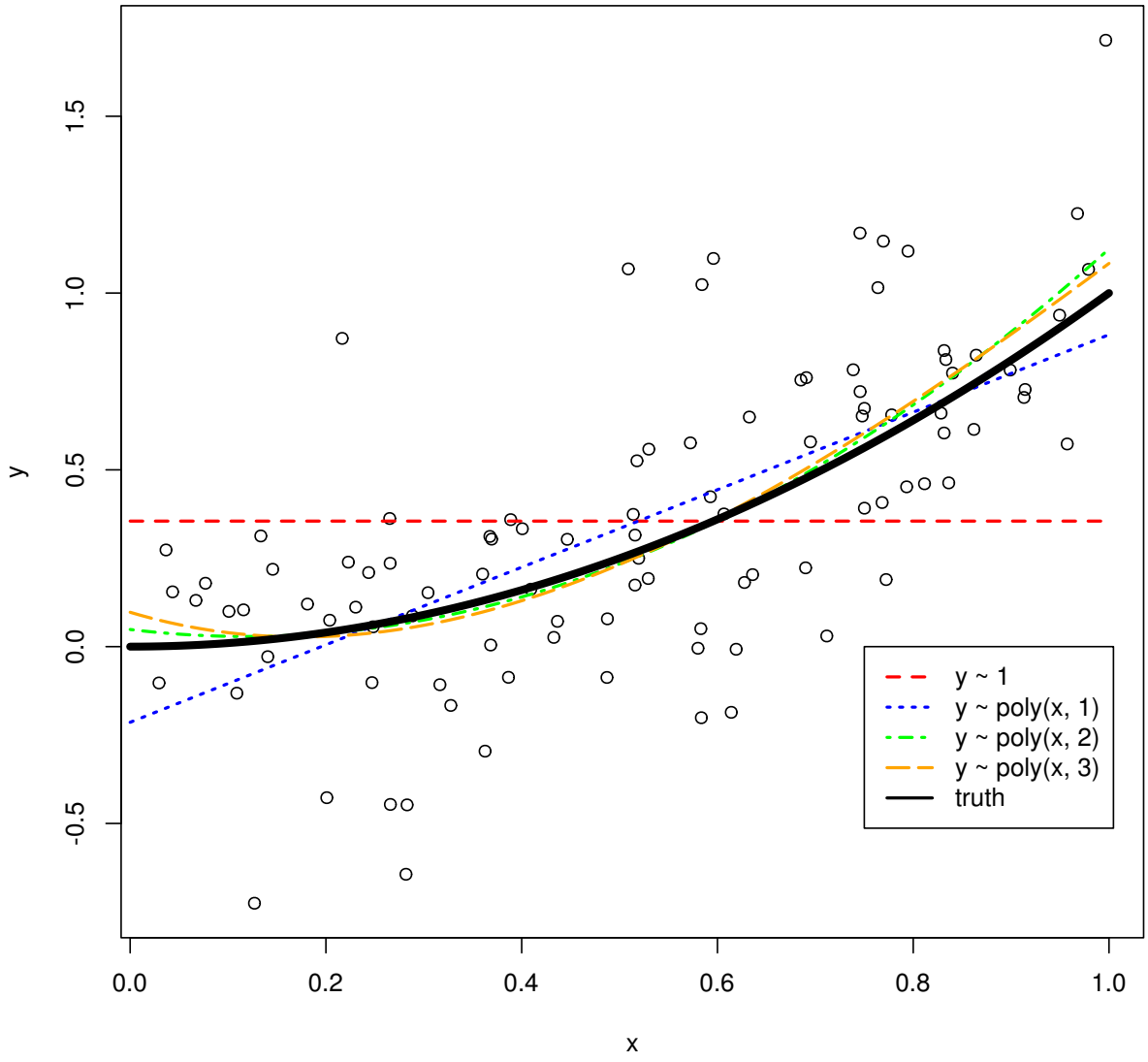
```
get_sim_data = function(f, sample_size = 100) {
  x = runif(n = sample_size, min = 0, max = 1)
  y = f(x) + rnorm(n = sample_size, mean = 0, sd = 0.3)
  data.frame(x, y)
}
```

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.

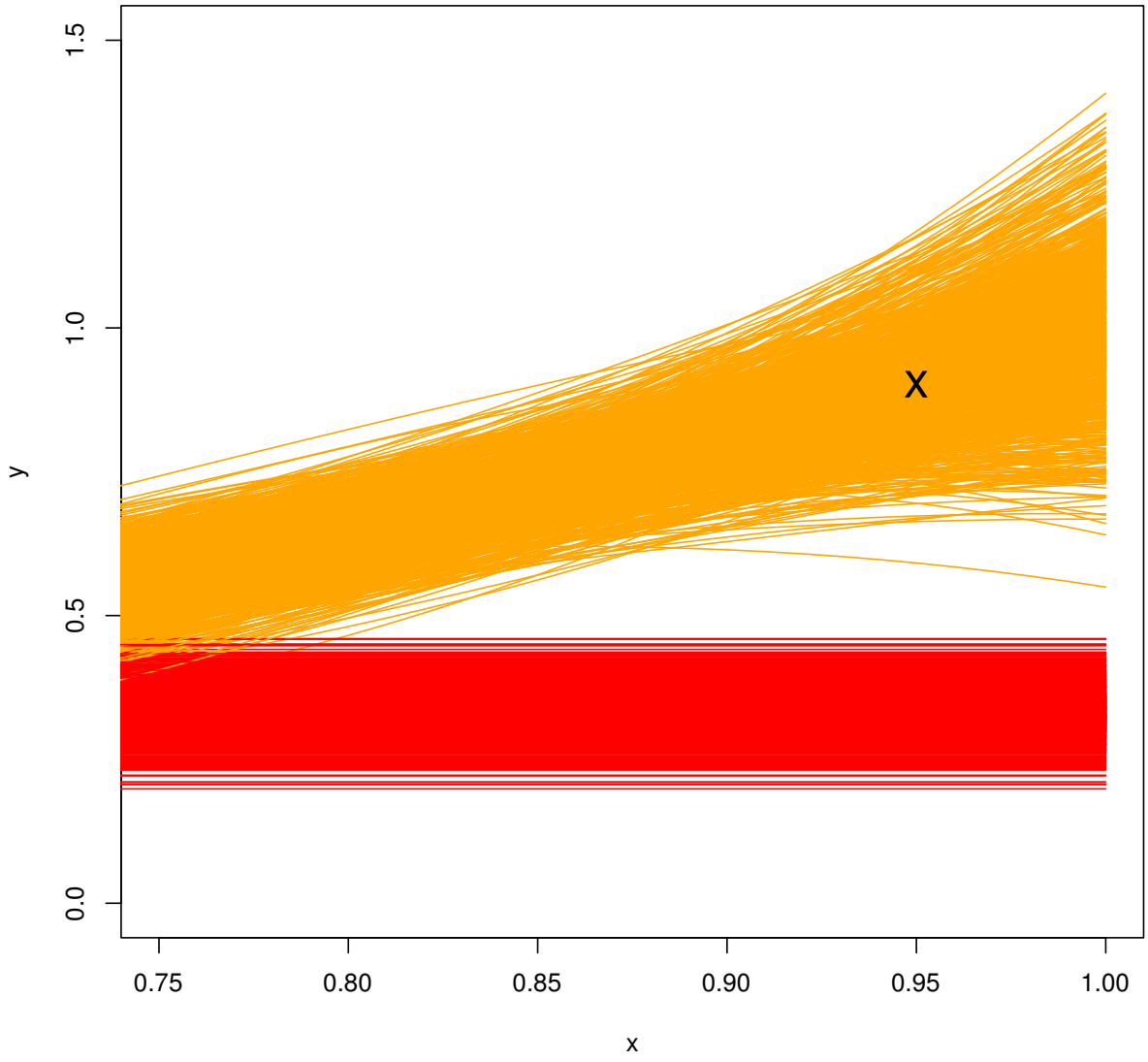
```
set.seed(430)
plot(y ~ x, data = sim_data)
grid = seq(from = 0, to = 1, by = 0.01)
lines(grid, predict(fit_1, newdata = data.frame(x = grid)),
      col = "red", lwd = 2, lty = 2)
lines(grid, predict(fit_2, newdata = data.frame(x = grid)),
      col = "blue", lwd = 2, lty = 3)
lines(grid, predict(fit_3, 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.

8.3 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 simulating 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
```

Part III

Classification

Chapter 9

Overview

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.*

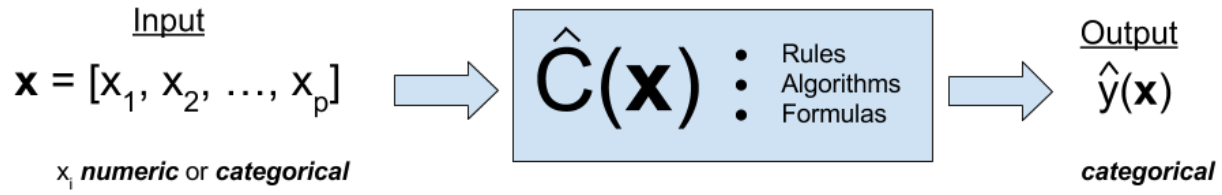


Figure 9.1:

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 x 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, ]
```

9.1 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.

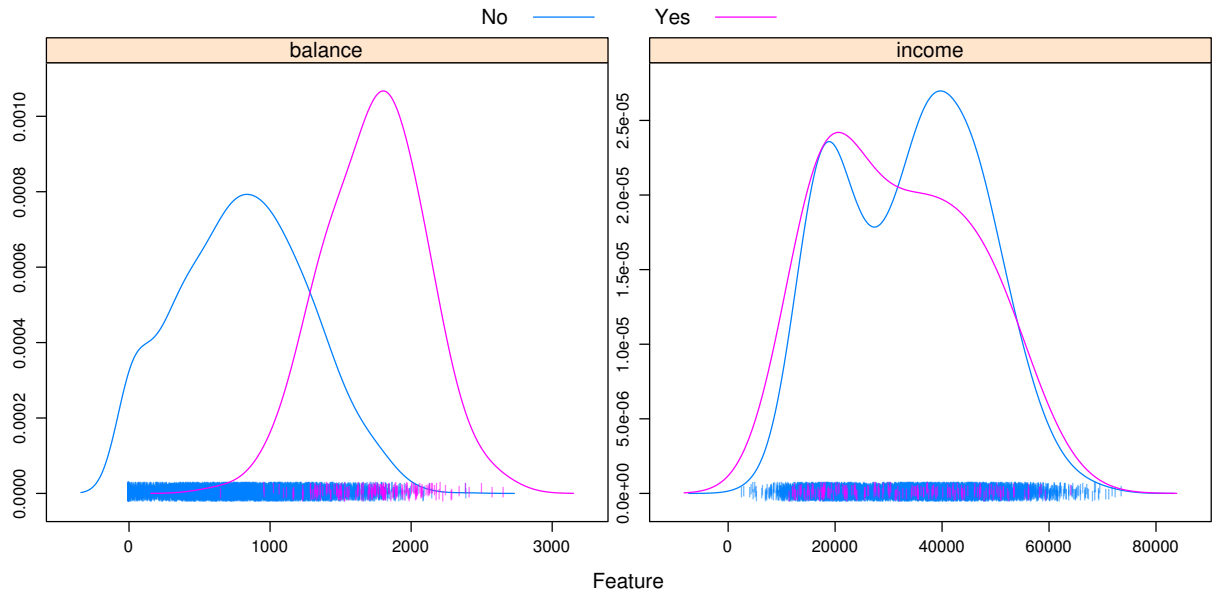
```
library(caret)
```

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))
```

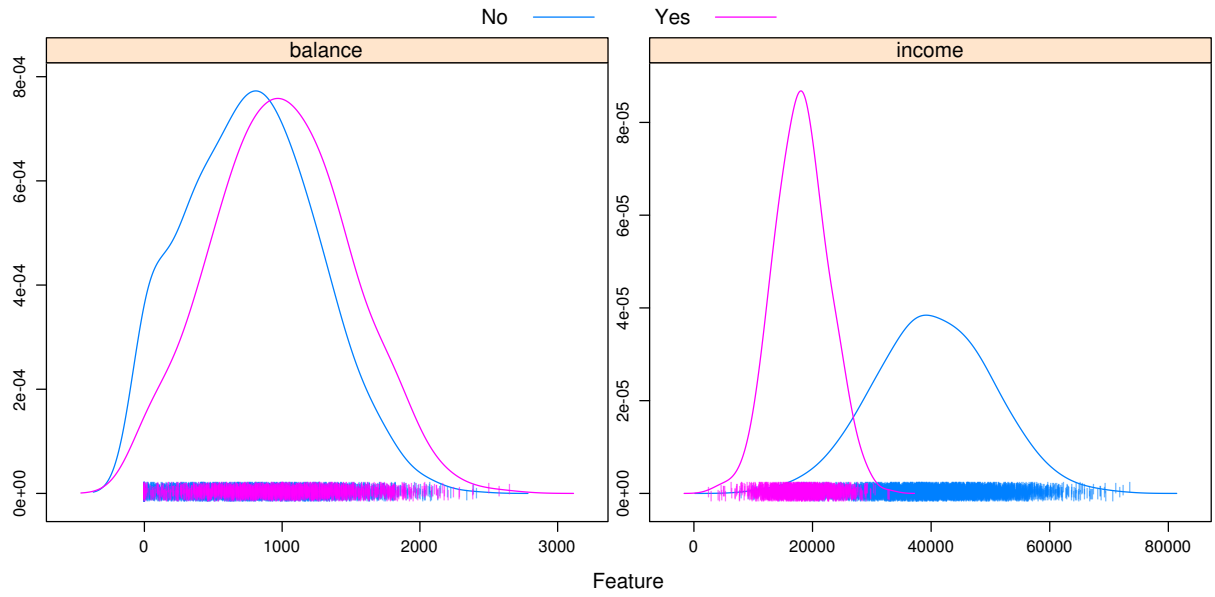


Some notes about the arguments to this function:

- `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.

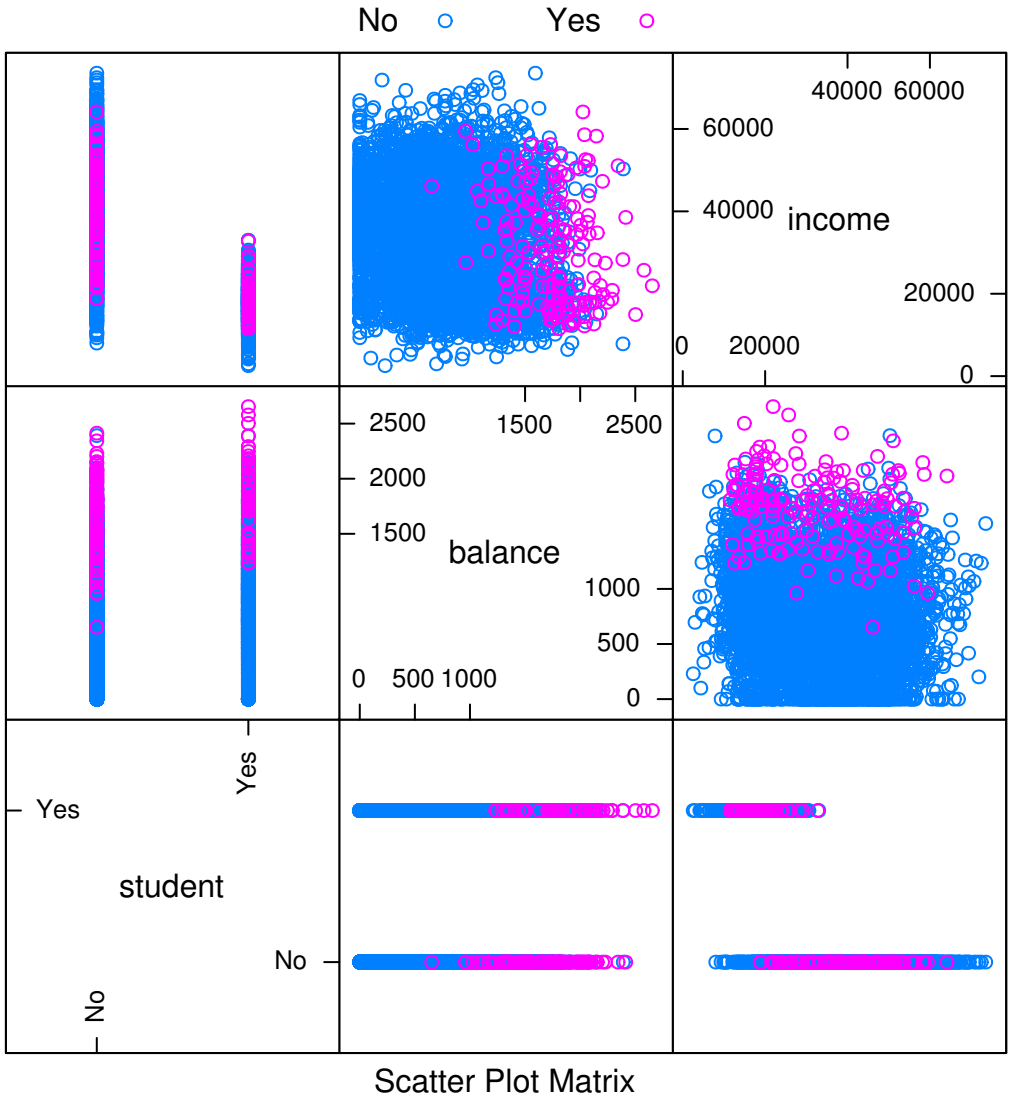
It seems that the income variable by itself is not particularly useful. However, there seems to be a big difference in default status at a `balance` of about 1400. We will use this information shortly.

```
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))
```



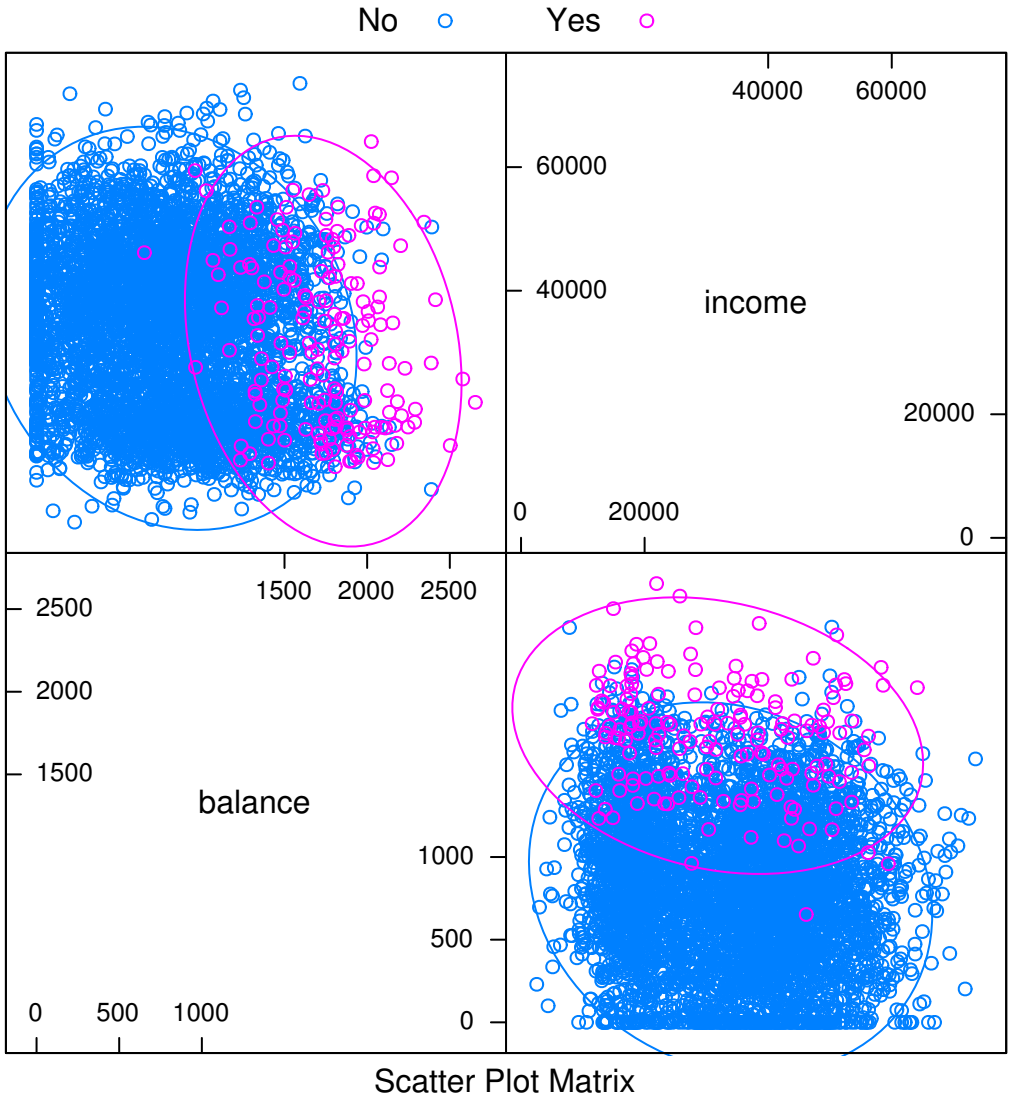
Above, we create a similar plot, except with `student` as the response. We see that students often carry a slightly larger balance, and have far lower income. This will be useful to know when making more complicated classifiers.

```
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.

9.2 A Simple Classifier

A very simple classifier is a rule based on a boundary b for a particular input variable x.

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & x > b \\ 0 & x \leq b \end{cases}$$

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, boundary, above = 1, below = 0) {
  ifelse(x > boundary, above, below)
}
```

We write a simple `R` function that compares a variable to a boundary, then use it to make predictions on the train and test sets with our chosen variable and boundary.

```
train_pred = simple_class(x = train_default$balance,
                          boundary = 1400, above = "Yes", below = "No")
test_pred = simple_class(x = test_default$balance,
                         boundary = 1400, above = "Yes", below = "No")
head(train_pred, n = 10)
```

```
##  [1] "No"  "Yes" "No"  "No"  "No"  "No"  "No"  "No"  "No"  "No"
```

9.3 Metrics for Classification

In the classification setting, there are a large number of metrics to assess 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 9.2:

```
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$balance,
                           boundary = 1400, above = "No", below = "No")
table(predicted = pred_all_no, actual = test_default$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 10

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 x 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, ]
```

10.1 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

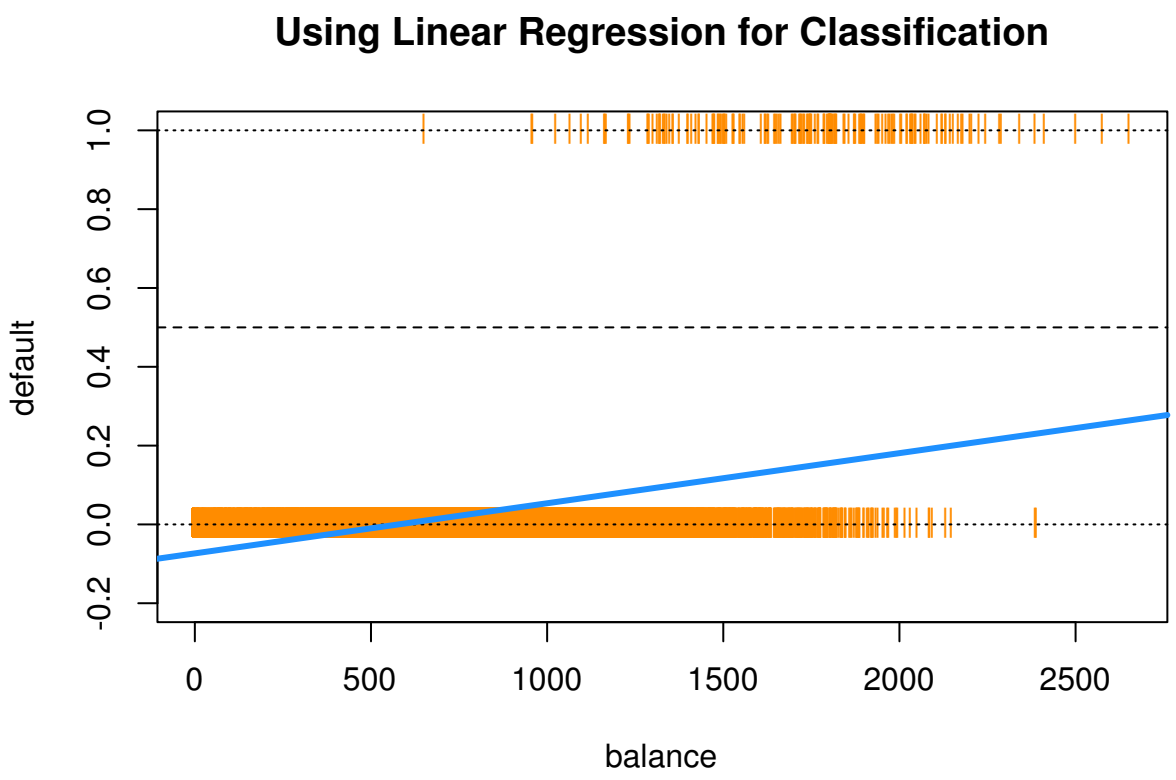
$$E[Y \mid X = x] = P[Y = 1 \mid X = x].$$

It would then seem reasonable that $X\hat{\beta}$ is a reasonable estimate of $P[Y = 1 \mid X = x]$. 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
```

10.2 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,

$$\hat{C}(\mathbf{x}) = \begin{cases} 1 & \hat{p}(\mathbf{x}) > 0.5 \\ 0 & \hat{p}(\mathbf{x}) \leq 0.5 \end{cases}$$

10.3 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 at 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.

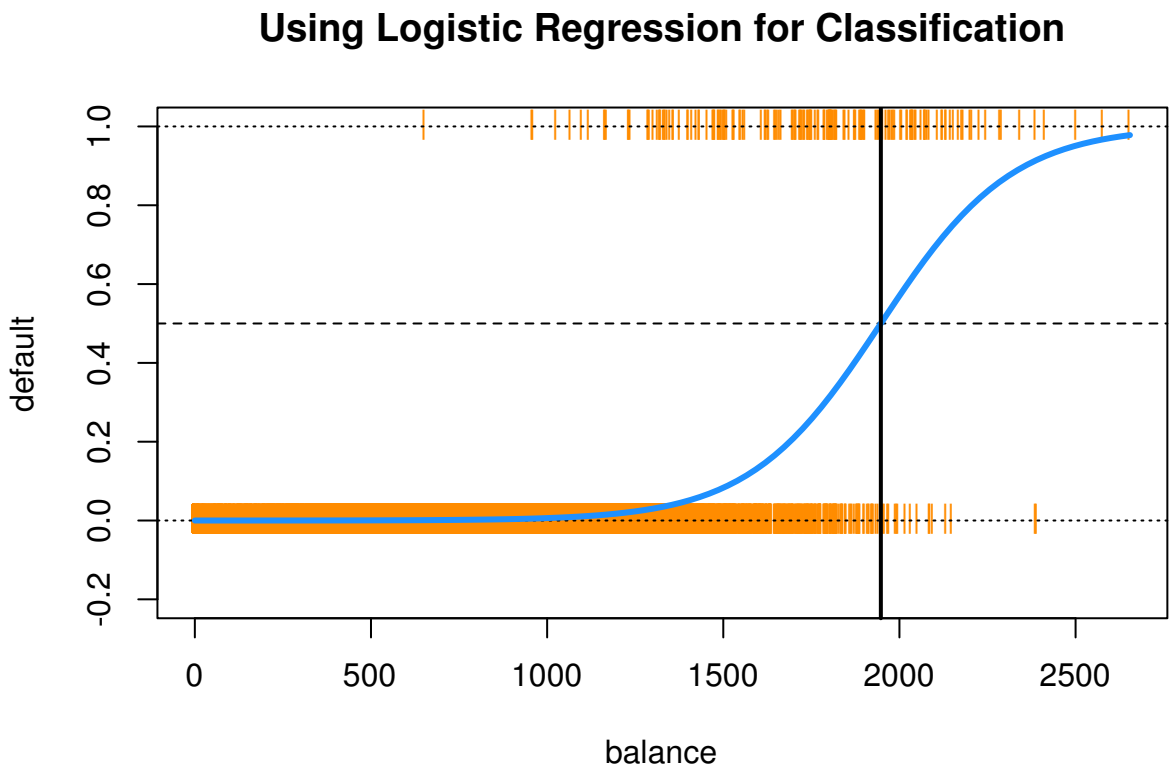
```
get_accuracy(model_glm, data = default_train,
             res = "default", pos = "Yes", neg = "No", cut = 0.5)
```

```
## [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.

10.4 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
```

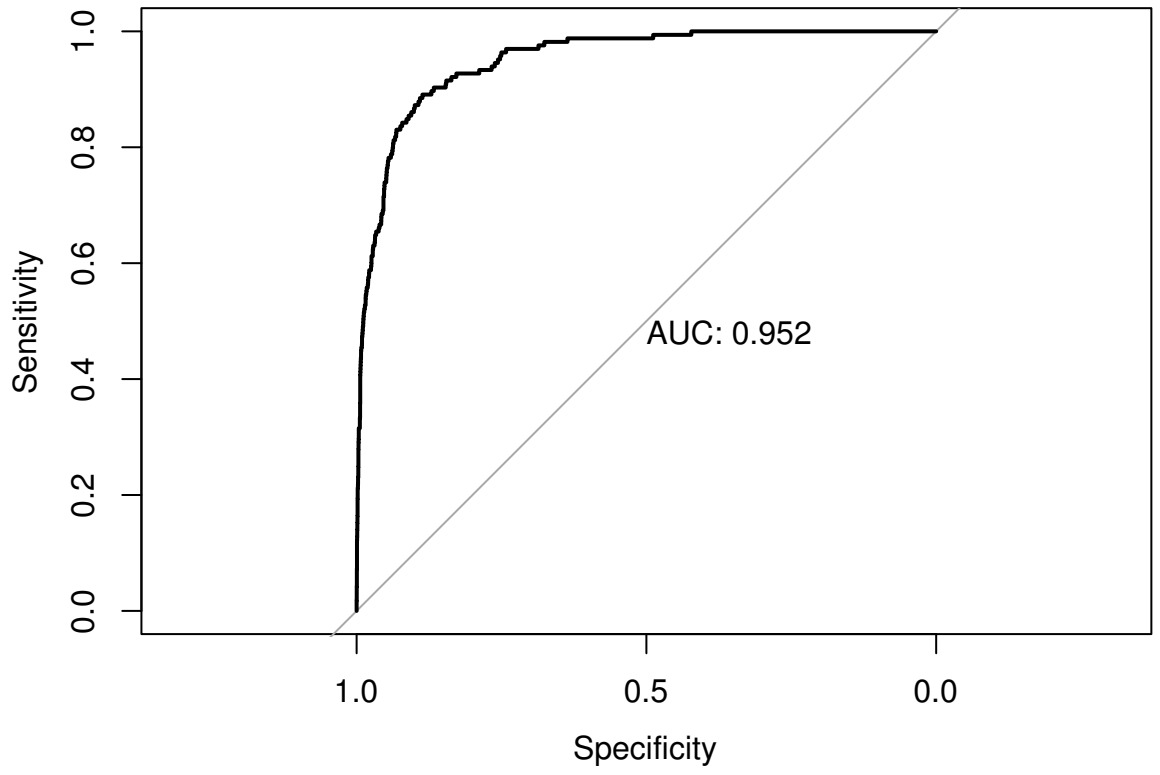
```
##          Accuracy Sensitivity Specificity
## c = 0.10   0.9404  0.77575758   0.9460186
## c = 0.50   0.9738  0.31515152   0.9962771
## c = 0.90   0.9674  0.01818182   0.9997932
```

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.

10.5 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 11

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

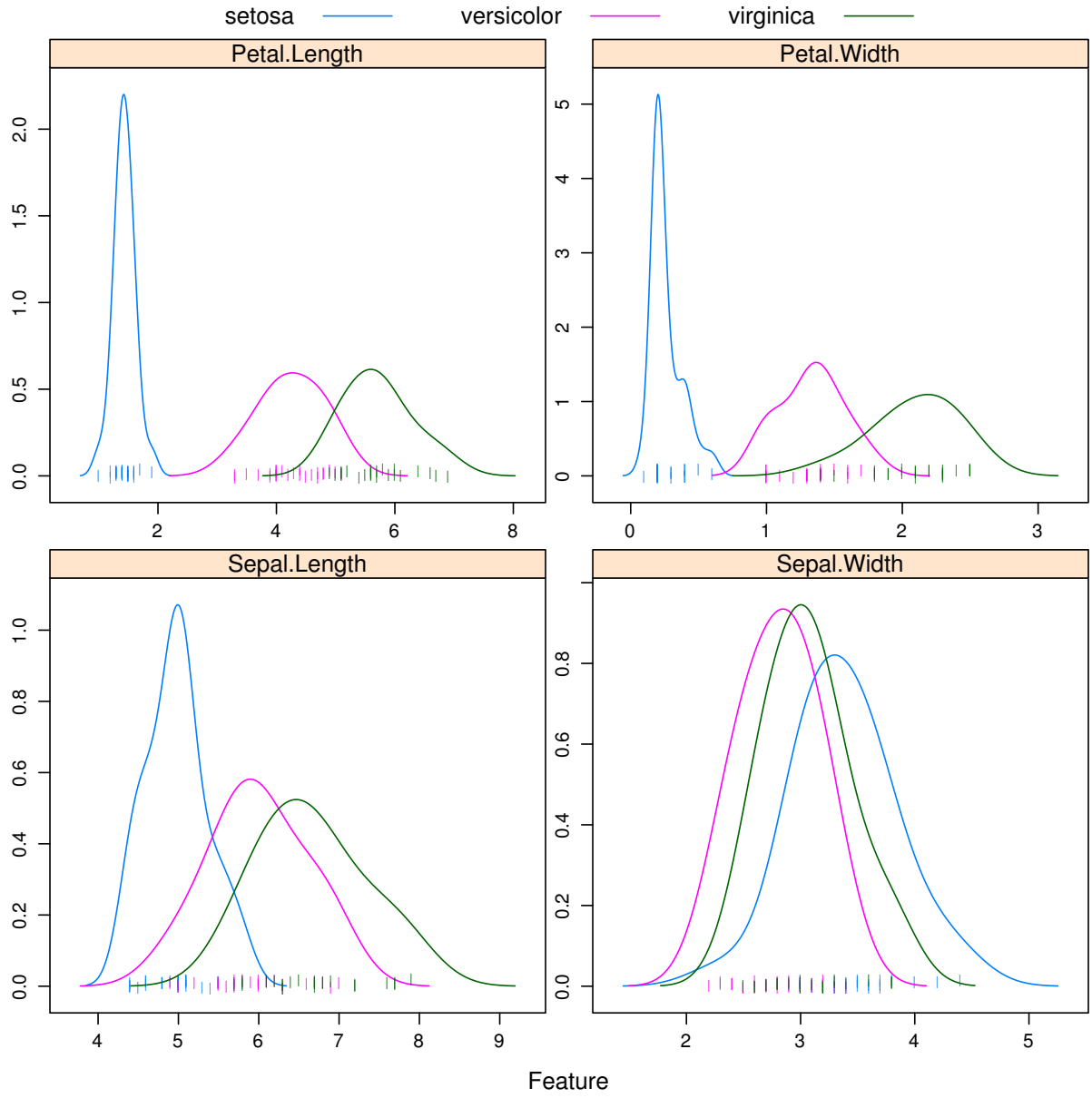
$$\log(\hat{\pi}_k \cdot \hat{f}_k(\mathbf{x})).$$

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.

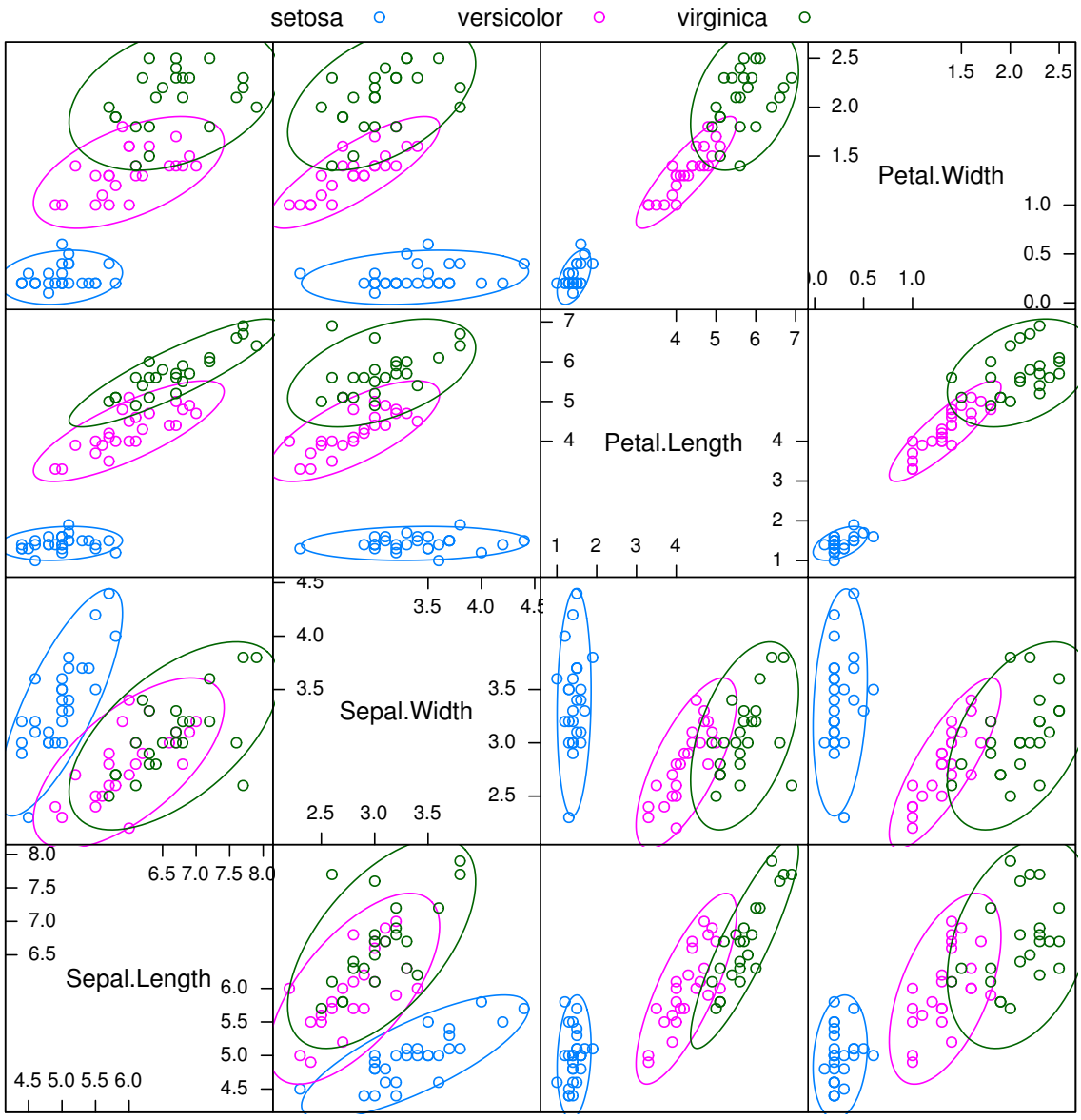
```
set.seed(430)
iris_obs = nrow(iris)
iris_index = sample(iris_obs, size = trunc(0.50 * iris_obs))
# iris_index = sample(iris_obs, size = trunc(0.10 * iris_obs))
iris_train = iris[iris_index, ]
iris_test = iris[-iris_index, ]
```

```
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))
```



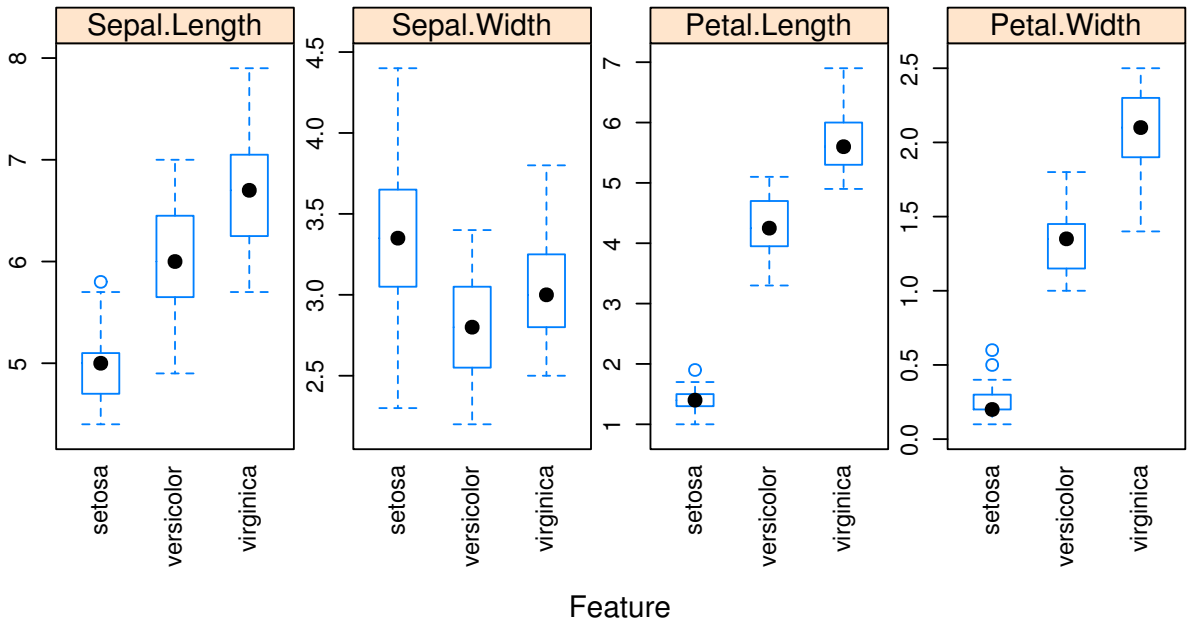

```
caret::featurePlot(x = iris_train[, c("Sepal.Length", "Sepal.Width",
                                      "Petal.Length", "Petal.Width")],
```

```
            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.

11.1 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!

11.2 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.

11.3 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.

11.4 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.

11.5 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
## [1] "e1071" "MASS"
```

- Additional Packages, Not Attached

```
##  [1] "reshape2"     "kernlab"      "purrr"        "splines"
##  [5] "lattice"      "colorspace"   "stats4"       "htmltools"
##  [9] "yaml"         "survival"     "prodlim"      "rlang"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "foreach"      "plyr"         "bindr"        "dimRed"
## [21] "lava"         "robustbase"   "stringr"      "timeDate"
## [25] "munsell"      "gtable"       "recipes"      "codetools"
## [29] "evaluate"     "knitr"        "caret"        "class"
## [33] "DEoptimR"     "methods"      "Rcpp"         "scales"
## [37] "backports"    "ipred"        "CVST"         "ellipse"
## [41] "ggplot2"      "digest"       "stringi"      "bookdown"
## [45] "dplyr"        "RcppRoll"     "ddalpha"      "grid"
## [49] "rprojroot"    "tools"        "magrittr"     "lazyeval"
## [53] "tibble"       "DRR"          "pkgconfig"    "Matrix"
## [57] "lubridate"    "gower"        "assertthat"   "rmarkdown"
## [61] "iterators"    "R6"           "rpart"        "nnet"
## [65] "nlme"         "compiler"
```

Chapter 12

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.

12.1 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.

12.1.1 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. Note that the y data should be a factor vector, **not** a data frame containing a factor vector.

```
# training data
X_default_train = default_train[, -1]
y_default_train = default_train$default

# testing data
X_default_test = default_test[, -1]
y_default_test = default_test$default
```

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
```

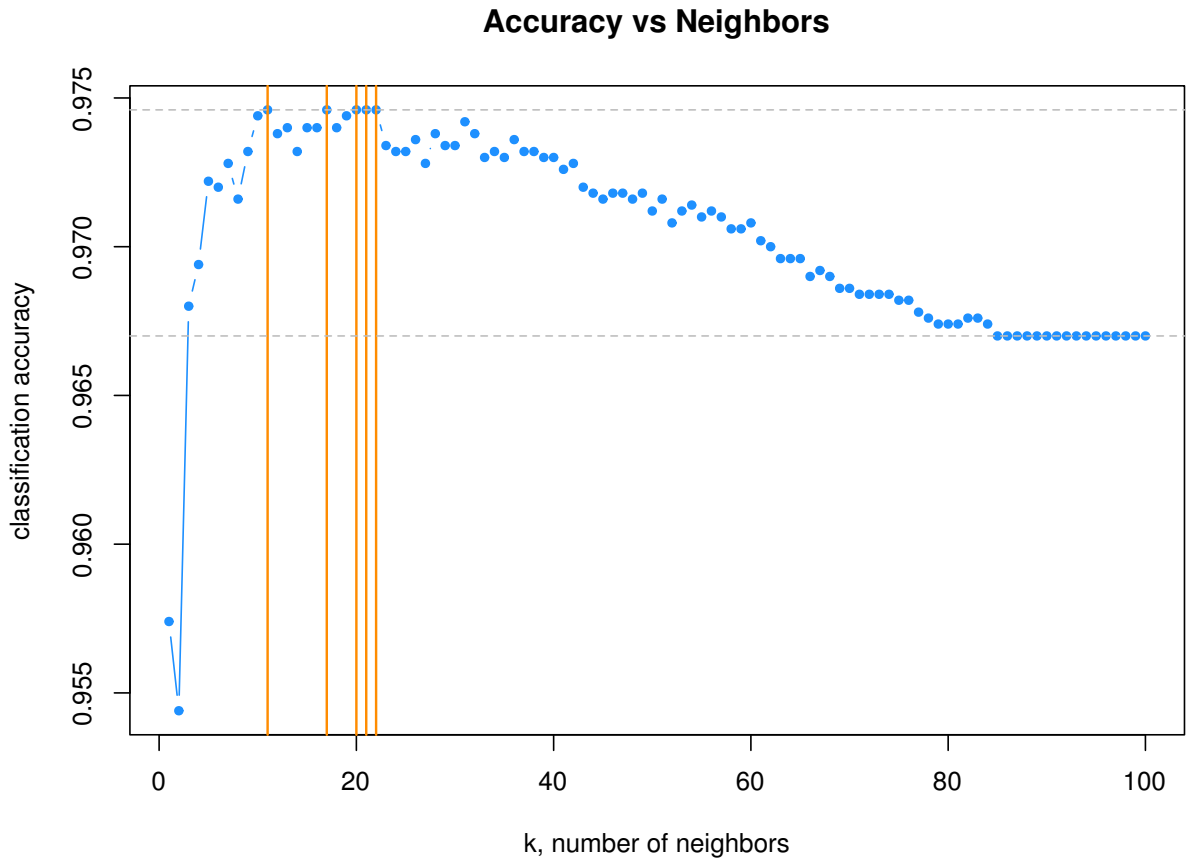
```
ex_storage = rep(x = 0, times = length(ex_seq))
for(i in seq_along(ex_seq)) {
  ex_storage[i] = mean(rnorm(n = 10, mean = ex_seq[i], sd = 1))
}

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
```

12.1.2 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$Species

# testing data
X_iris_test = iris_test[, -5]
y_iris_test = iris_test$Species
```

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
```

12.2 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 we won't be checking RMSE, but instead plotting fitted models. To make plotting easier, we won't scale the data.

```
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))
```

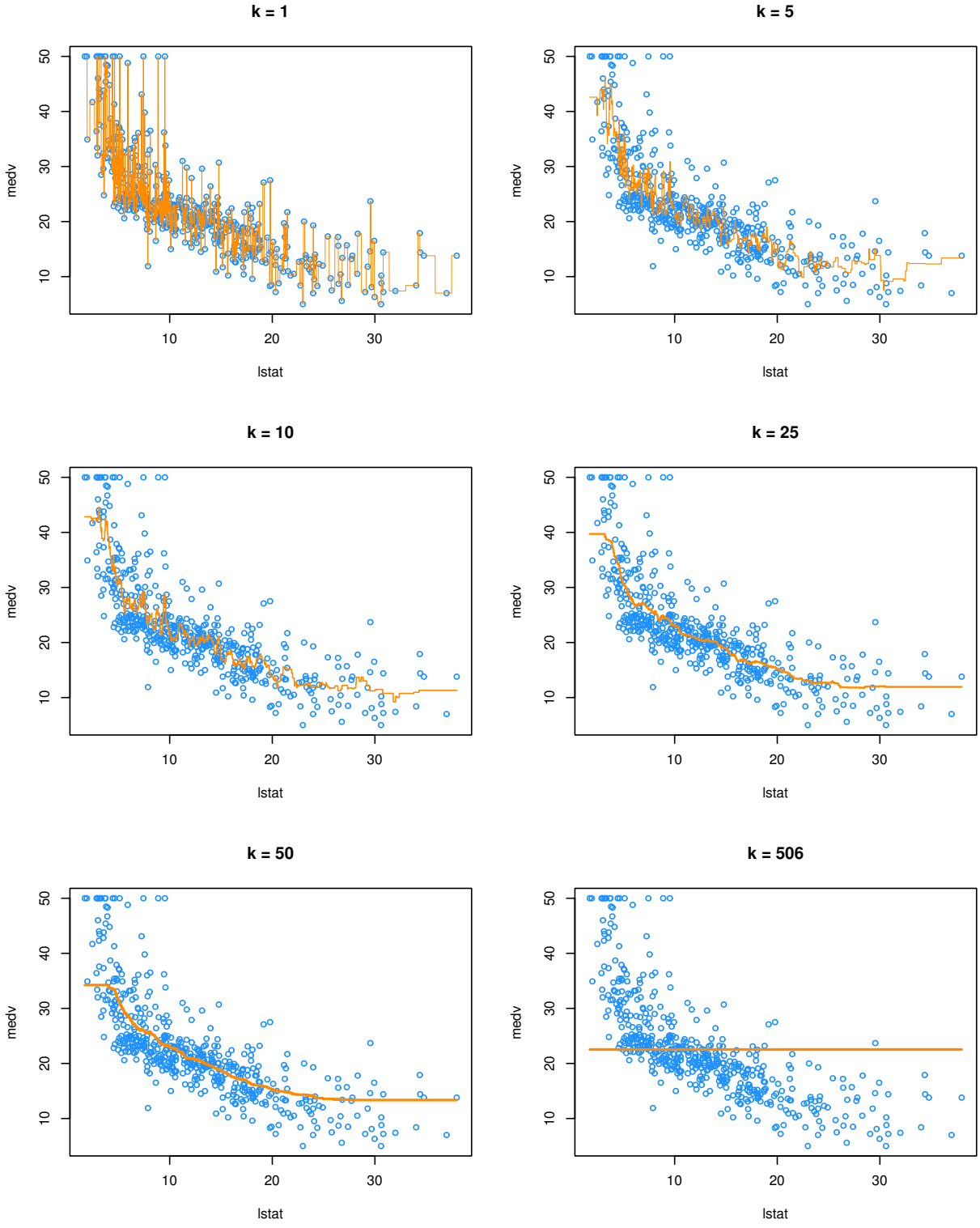
Unfortunately, `knn()` from `class` only handles classification. To perform regression, we will need `knn.reg()` from the `FNN` package. Notice that, we do **not** load this package, but instead use `FNN::knn.reg` to access the function. This is useful since `FNN` also contains a function `knn()` and 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 total 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.

12.3 External Links

- YouTube: k-Nearest Neighbor Classification Algorithm - Video from user "mathematicalmonk" which gives a brief but thorough introduction to the method.

12.4 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
## [1] "MASS"  "class" "ISLR"
```

- Additional Packages, Not Attached

```
##  [1] "Rcpp"      "bookdown"  "FNN"       "digest"    "rprojroot"
##  [6] "backports" "magrittr"  "evaluate"  "stringi"   "rmarkdown"
## [11] "tools"     "stringr"   "yaml"      "compiler"  "htmltools"
## [16] "knitr"     "methods"
```

Part IV

Unsupervised Learning

Chapter 13

Overview

TODO: Move current content into the following placeholder chapters. Add details.

13.1 Methods

13.1.1 Principal Component Analysis

To perform PCA in `R` we will use `prcomp()`. See `?prcomp()` for details.

13.1.2 k-Means Clustering

To perform k-means in `R` we will use `kmeans()`. See `?kmeans()` for details.

13.1.3 Hierarchical Clustering

To perform hierarchical clustering in `R` we will use `hclust()`. See `?hclust()` for details.

13.2 Examples

13.2.1 US Arrests

```
library(ISLR)
data(USArrests)
apply(USArrests, 2, mean)
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

```
apply(USArrests, 2, sd)
```

```
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

"Before" performing PCA, we will scale the data. (This will actually happen inside the `prcomp()` function.)

```
USArrests_pca = prcomp(USArrests, scale = TRUE)
```

A large amount of information is stored in the output of `prcomp()`, some of which can neatly be displayed with `summary()`.

```
names(USArrests_pca)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

```
summary(USArrests_pca)
```

```
## Importance of components%s:
##                           PC1    PC2     PC3     PC4
## Standard deviation     1.5749 0.9949 0.59713 0.41645
## Proportion of Variance 0.6201 0.2474 0.08914 0.04336
## Cumulative Proportion  0.6201 0.8675 0.95664 1.00000
```

```
USArrests_pca$center
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

```
USArrests_pca$scale
```

```
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

```
USArrests_pca$rotation
```

```
##                 PC1        PC2        PC3         PC4
## Murder   -0.5358995  0.4181809 -0.3412327  0.64922780
## Assault  -0.5831836  0.1879856 -0.2681484 -0.74340748
## UrbanPop -0.2781909 -0.8728062 -0.3780158  0.13387773
## Rape     -0.5434321 -0.1673186  0.8177779  0.08902432
```

We see that `$center` and `$scale` give the mean and standard deviations for the original variables. `$rotation` gives the loading vectors that are used to rotate the original data to obtain the principal components.

```
dim(USArrests_pca$x)
```

```
## [1] 50  4
```

```
dim(USArrests)
```

```
## [1] 50  4
```

```
head(USArrests_pca$x)
```

```
##                   PC1        PC2         PC3          PC4
## Alabama    -0.9756604  1.1220012 -0.43980366  0.154696581
## Alaska     -1.9305379  1.0624269  2.01950027 -0.434175454
## Arizona    -1.7454429 -0.7384595  0.05423025 -0.826264240
## Arkansas    0.1399989  1.1085423  0.11342217 -0.180973554
## California -2.4986128 -1.5274267  0.59254100 -0.338559240
## Colorado   -1.4993407 -0.9776297  1.08400162  0.001450164
```

The dimension of the principal components is the same as the original data. The principal components are stored in $x.

```
scale(as.matrix(USArrests))[1, ] %*% USArrests_pca$rotation[, 1]
```

```
##            [,1]
## [1,] -0.9756604
```

```
scale(as.matrix(USArrests))[1, ] %*% USArrests_pca$rotation[, 2]
```

```
##          [,1]
## [1,] 1.122001
```

```
scale(as.matrix(USArrests))[1, ] %*% USArrests_pca$rotation[, 3]
```

```
##            [,1]
## [1,] -0.4398037
```

```
scale(as.matrix(USArrests))[1, ] %*% USArrests_pca$rotation[, 4]
```

```
##           [,1]
## [1,] 0.1546966
```

```
head(scale(as.matrix(USArrests)) %*% USArrests_pca$rotation[,1])
```

```
##                  [,1]
## Alabama    -0.9756604
## Alaska     -1.9305379
## Arizona    -1.7454429
## Arkansas    0.1399989
## California -2.4986128
## Colorado   -1.4993407
```

```
head(USArrests_pca$x[, 1])
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
## -0.9756604 -1.9305379 -1.7454429  0.1399989 -2.4986128 -1.4993407
```

```
sum(USArrests_pca$rotation[, 1] ^ 2)
```

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

```
USArrests_pca$rotation[, 1] %*% USArrests_pca$rotation[, 2]
```

```
##                [,1]
## [1,] -1.387779e-16
```

```
USArrests_pca$rotation[, 1] %*% USArrests_pca$rotation[, 3]
```

```
##                [,1]
## [1,] -5.551115e-17
```

```
USArrests_pca$x[, 1] %*% USArrests_pca$x[, 2]
```

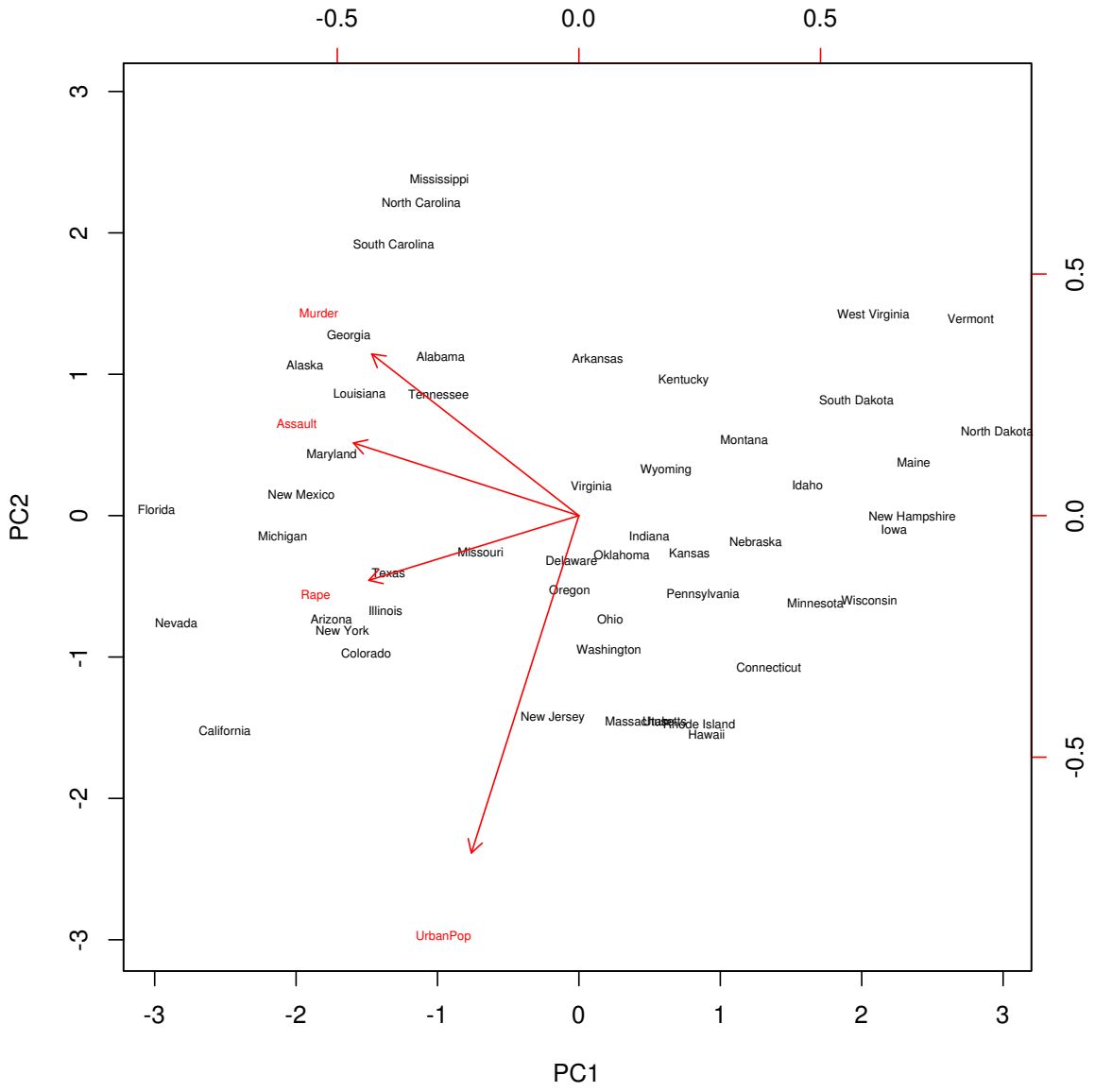
```
##                [,1]
## [1,] -2.062239e-14
```

```
USArrests_pca$x[, 1] %*% USArrests_pca$x[, 3]
```

```
##               [,1]
## [1,] 5.384582e-15
```

The above verifies some of the "math" of PCA. We see how the loadings obtain the principal components from the original data. We check that the loading vectors are normalized. We also check for orthogonality of both the loading vectors and the principal components. (Note the above inner products aren't exactly 0, but that is simply a numerical issue.)

```
biplot(USArrests_pca, scale = 0, cex = 0.5)
```



A `biplot` can be used to visualize both the principal component scores and the principal component loadings. (Note the two scales for each axis.)

```
USArrests_pca$sdev
```

```
## [1] 1.5748783 0.9948694 0.5971291 0.4164494
```

```
USArrests_pca$sdev ^ 2 / sum(USArrests_pca$sdev ^ 2)
```

```
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

Frequently we will be interested in the proportion of variance explained by a principal component.

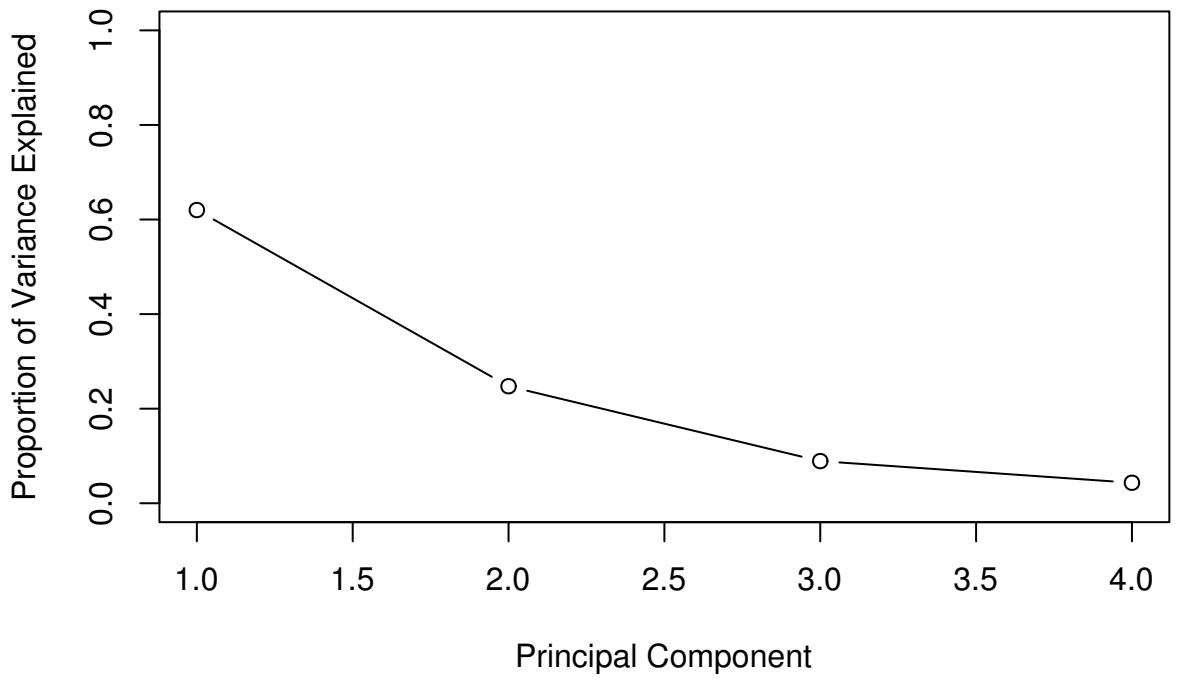
```
get_PVE = function(pca_out) {
  pca_out$sdev ^ 2 / sum(pca_out$sdev ^ 2)
}
```

So frequently, we would be smart to write a function to do so.

```
pve = get_PVE(USArrests_pca)

pve
```

```
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

```
plot(
  pve,
  xlab = "Principal Component",
  ylab = "Proportion of Variance Explained",
  ylim = c(0, 1),
  type = 'b'
)
```

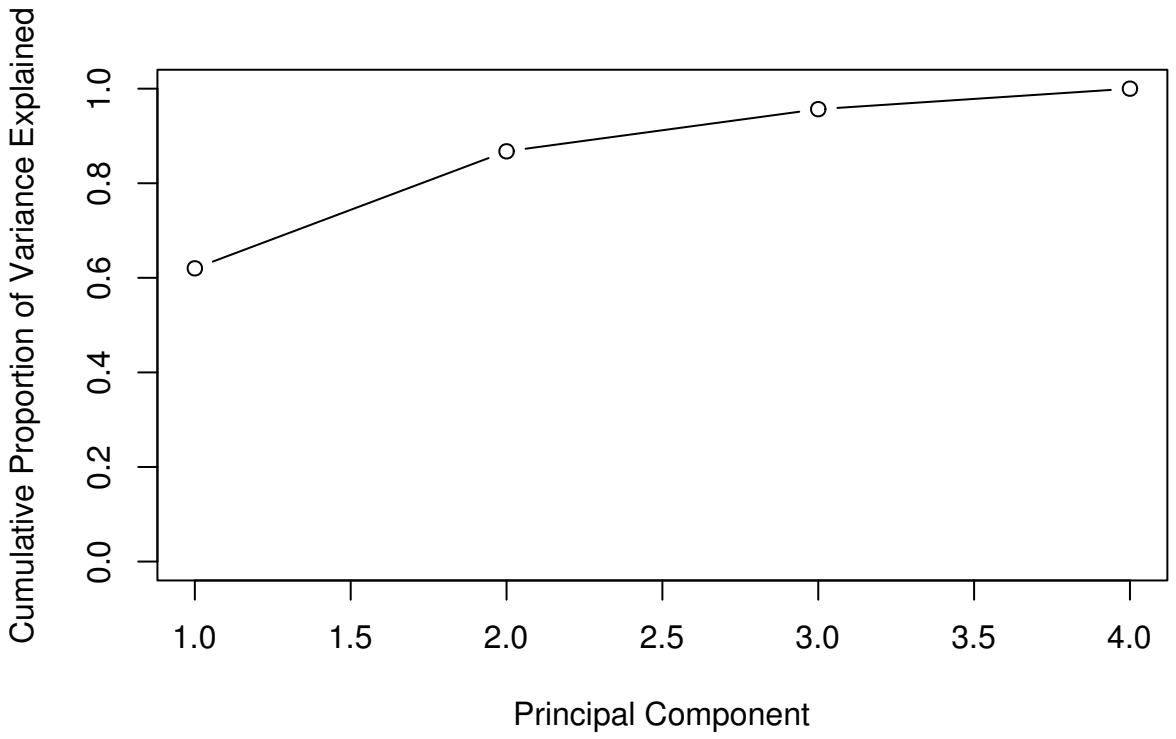


We can then plot the proportion of variance explained for each PC. As expected, we see the PVE decrease.

```
cumsum(pve)
```

```
## [1] 0.6200604 0.8675017 0.9566425 1.0000000
```

```
plot(
  cumsum(pve),
  xlab = "Principal Component",
  ylab = "Cumulative Proportion of Variance Explained",
  ylim = c(0, 1),
  type = 'b'
)
```



Often we are interested in the cumulative proportion. A common use of PCA outside of visualization is dimension reduction for modeling. If p is large, PCA is performed, and the principal components that account for a large proportion of variation, say 95%, are used for further analysis. In certain situations that can reduce the dimensionality of data significantly. This can be done almost automatically using `caret`:

```
library(caret)
library(mlbench)
data(Sonar)
set.seed(18)
using_pca = train(Class ~ ., data = Sonar, method = "knn",
                  trControl = trainControl(method = "cv", number = 5),
                  preProcess = "pca",
                  tuneGrid = expand.grid(k = c(1, 3, 5, 7, 9)))
regular_scaling = train(Class ~ ., data = Sonar, method = "knn",
                        trControl = trainControl(method = "cv", number = 5),
                        preProcess = c("center", "scale"),
                        tuneGrid = expand.grid(k = c(1, 3, 5, 7, 9)))
max(using_pca$results$Accuracy)
```

```
## [1] 0.8656997
```

```
max(regular_scaling$results$Accuracy)
```

```
## [1] 0.8609378
```

```
using_pca$preProcess
```

```
## Created from 208 samples and 60 variables
##
## Pre-processing:
##   - centered (60)
##   - ignored (0)
##   - principal component signal extraction (60)
##   - scaled (60)
##
## PCA needed 30 components to capture 95 percent of the variance
```

It won't always outperform simply using the original predictors, but here using 30 of 60 principal components shows a slight advantage over using all 60 predictors. In other situation, it may result in a large perform gain.

13.2.2 Simulated Data

```
library(MASS)
set.seed(42)
n = 180
p = 10
clust_data = rbind(
  mvrnorm(n = n / 3, sample(c(1, 2, 3, 4), p, replace = TRUE), diag(p)),
  mvrnorm(n = n / 3, sample(c(1, 2, 3, 4), p, replace = TRUE), diag(p)),
  mvrnorm(n = n / 3, sample(c(1, 2, 3, 4), p, replace = TRUE), diag(p))
)
```

Above we simulate data for clustering. Note that, we did this in a way that will result in three clusters.

```
true_clusters = c(rep(3, n / 3), rep(1, n / 3), rep(2, n / 3))
```

We label the true clusters 1, 2, and 3 in a way that will "match" output from k-means. (Which is somewhat arbitrary.)

```
kmean_out = kmeans(clust_data, centers = 3, nstart = 10)
names(kmean_out)
```

```
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

Notice that we used `nstart = 10` which will give us a more stable solution by attempting 10 random starting positions for the means. Also notice we chose to use `centers = 3`. (The k in k-mean). How did we know to do this? We'll find out on the homework. (It will involve looking at `tot.withinss`)

```
kmean_out
```

```
## K-means clustering with 3 clusters of sizes 61, 60, 59
##
## Cluster means:
##       [,1]     [,2]      [,3]     [,4]     [,5]      [,6]     [,7]
## 1 3.997352 4.085592 0.7846534 2.136643 4.059886 3.2490887 1.747697
## 2 1.008138 2.881229 4.3102354 4.094867 3.022989 0.8878413 4.002270
## 3 3.993468 4.049505 1.9553560 4.037748 2.825907 2.9960855 3.026397
##        [,8]      [,9]    [,10]
## 1 1.8341976 0.8193371 4.043725
## 2 3.8085492 2.0905060 0.977065
## 3 0.8992179 3.0041820 2.931030
##
## Clustering vector:
##   [1] 3 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [36] 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [141] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [176] 2 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1] 609.2674 581.8780 568.3845
##  (between_SS / total_SS =  54.0 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

```
kmeans_clusters = kmean_out$cluster

table(true_clusters, kmeans_clusters)
```

```
##              kmeans_clusters
## true_clusters  1  2  3
##             1 58  0  2
##             2  0 60  0
##             3  3  0 57
```

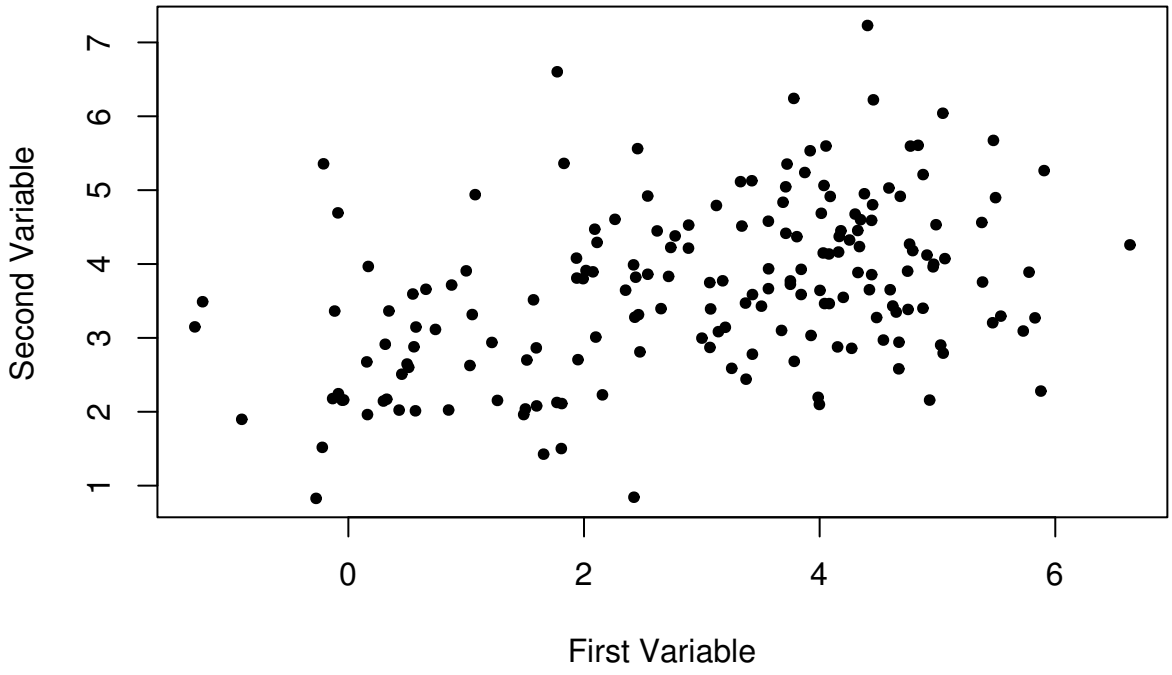
We check how well the clustering is working.

```
dim(clust_data)
```

```
## [1] 180  10
```

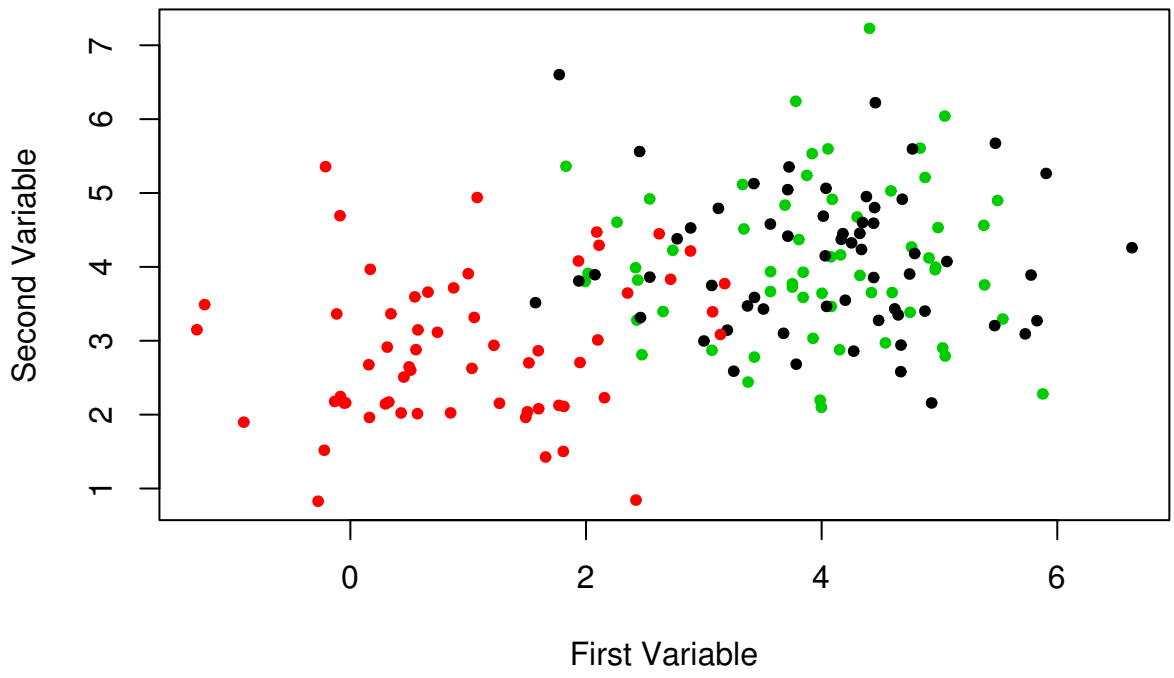
This data is "high dimensional" so it is difficult to visualize. (Anything more than 2 is hard to visualize.)

```
plot(
  clust_data[, 1],
  clust_data[, 2],
  pch = 20,
  xlab = "First Variable",
  ylab = "Second Variable"
)
```



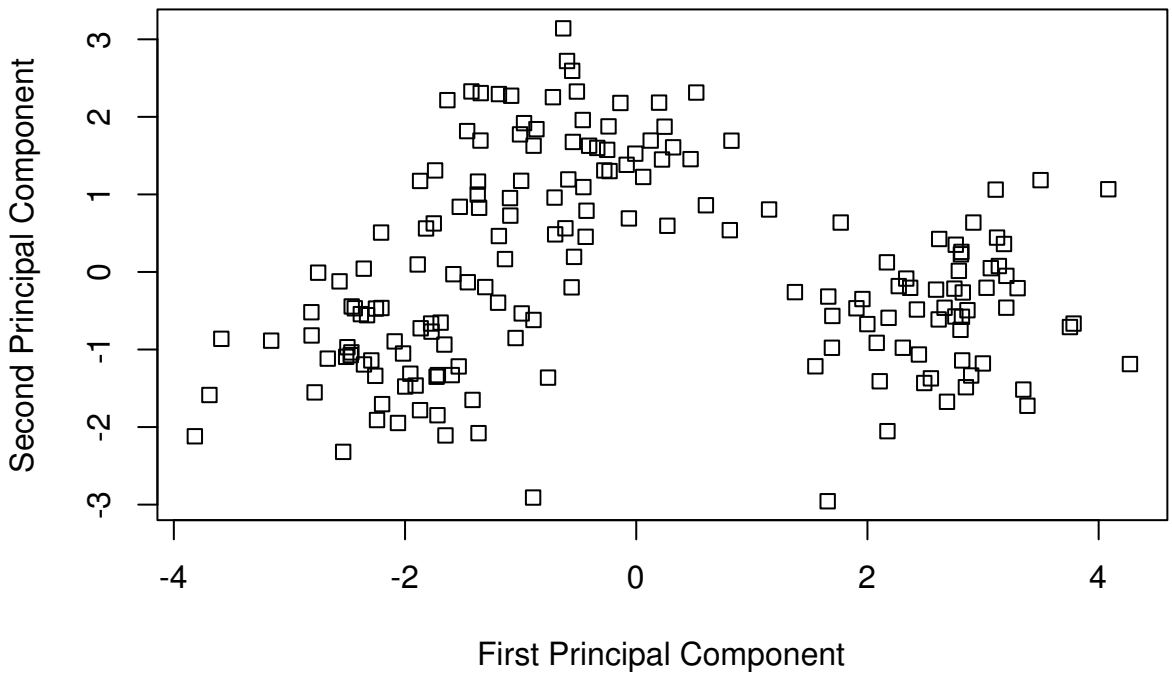
Plotting the first and second variables simply results in a blob.

```
plot(
  clust_data[, 1],
  clust_data[, 2],
  col = true_clusters,
  pch = 20,
  xlab = "First Variable",
  ylab = "Second Variable"
)
```



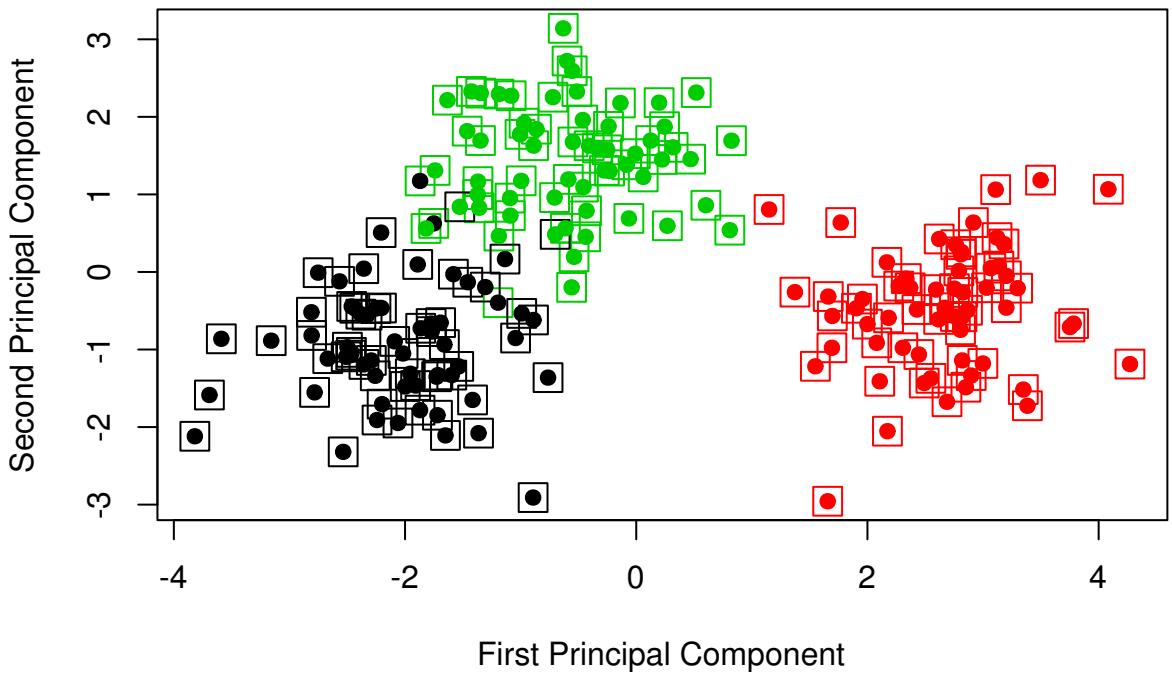
Even when using their true clusters for coloring, this plot isn't very helpful.

```
clust_data_pca = prcomp(clust_data, scale = TRUE)

plot(
  clust_data_pca$x[, 1],
  clust_data_pca$x[, 2],
  pch = 0,
  xlab = "First Principal Component",
  ylab = "Second Principal Component"
)
```



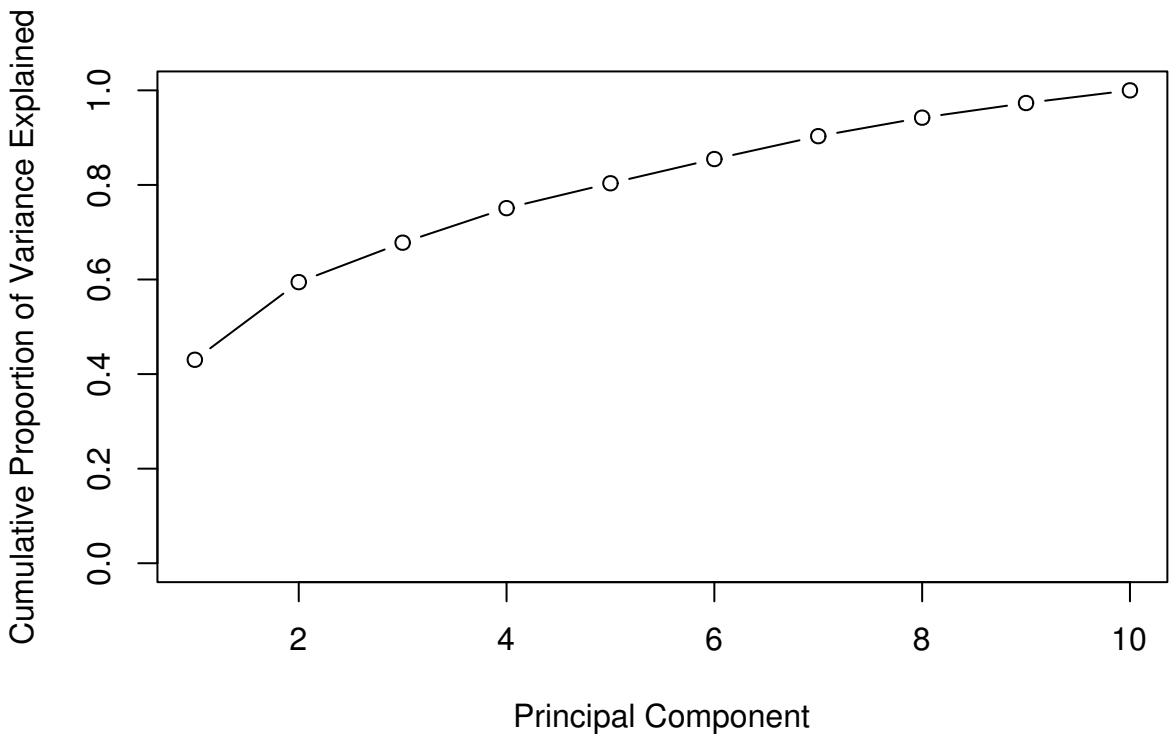
If we instead plot the first two principal components, we see, even without coloring, one blob that is clearly separate from the rest.

```
plot(
  clust_data_pca$x[, 1],
  clust_data_pca$x[, 2],
  col = true_clusters,
  pch = 0,
  xlab = "First Principal Component",
  ylab = "Second Principal Component",
  cex = 2
)
points(clust_data_pca$x[, 1], clust_data_pca$x[, 2], col = kmeans_clusters, pch = 20, cex = 1.5)
```



Now adding the true colors (boxes) and the k-means results (circles), we obtain a nice visualization.

```
clust_data_pve = get_PVE(clust_data_pca)

plot(
  cumsum(clust_data_pve),
  xlab = "Principal Component",
  ylab = "Cumulative Proportion of Variance Explained",
  ylim = c(0, 1),
  type = 'b'
)
```

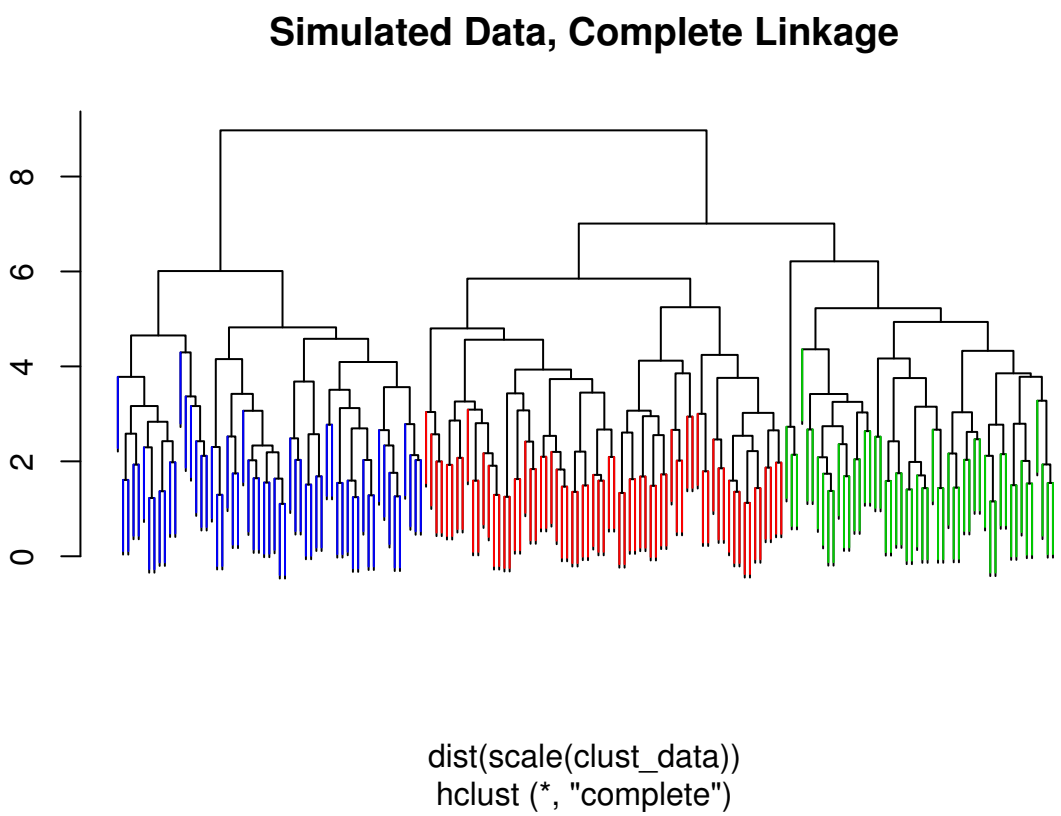


The above visualization works well because the first two PCs explain a large proportion of the variance.

```
#install.packages('sparcl')
library(sparcl)
```

To create colored dendrograms we will use `ColorDendrogram()` in the `sparcl` package.

```
clust_data_hc = hclust(dist(scale(clust_data)), method = "complete")
clust_data_cut = cutree(clust_data_hc , 3)
ColorDendrogram(clust_data_hc, y = clust_data_cut,
                labels = names(clust_data_cut),
                main = "Simulated Data, Complete Linkage",
                branchlength = 1.5)
```



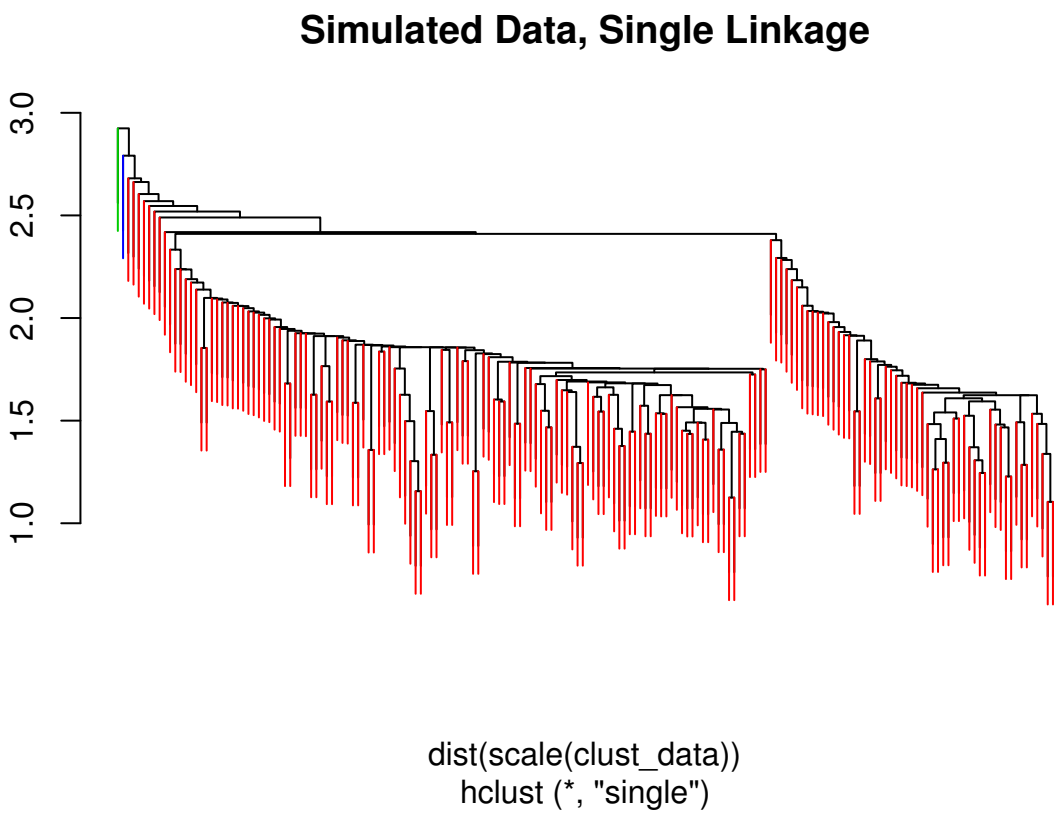
Here we apply hierarchical clustering to the **scaled** data. The `dist()` function is used to calculate pairwise distances between the (scaled in this case) observations. We use complete linkage. We then use the `cutree()` function to cluster the data into `3` clusters. The `ColorDendrogram()` function is then used to plot the dendrogram. Note that the `branchlength` argument is somewhat arbitrary (the length of the colored bar) and will need to be modified for each dendrogram.

```
table(true_clusters, clust_data_cut)
```

```
##              clust_data_cut
## true_clusters  1  2  3
##             1  9 51  0
##             2  1  0 59
##             3 59  1  0
```

We see in this case hierarchical clustering doesn't "work" as well as k-means.

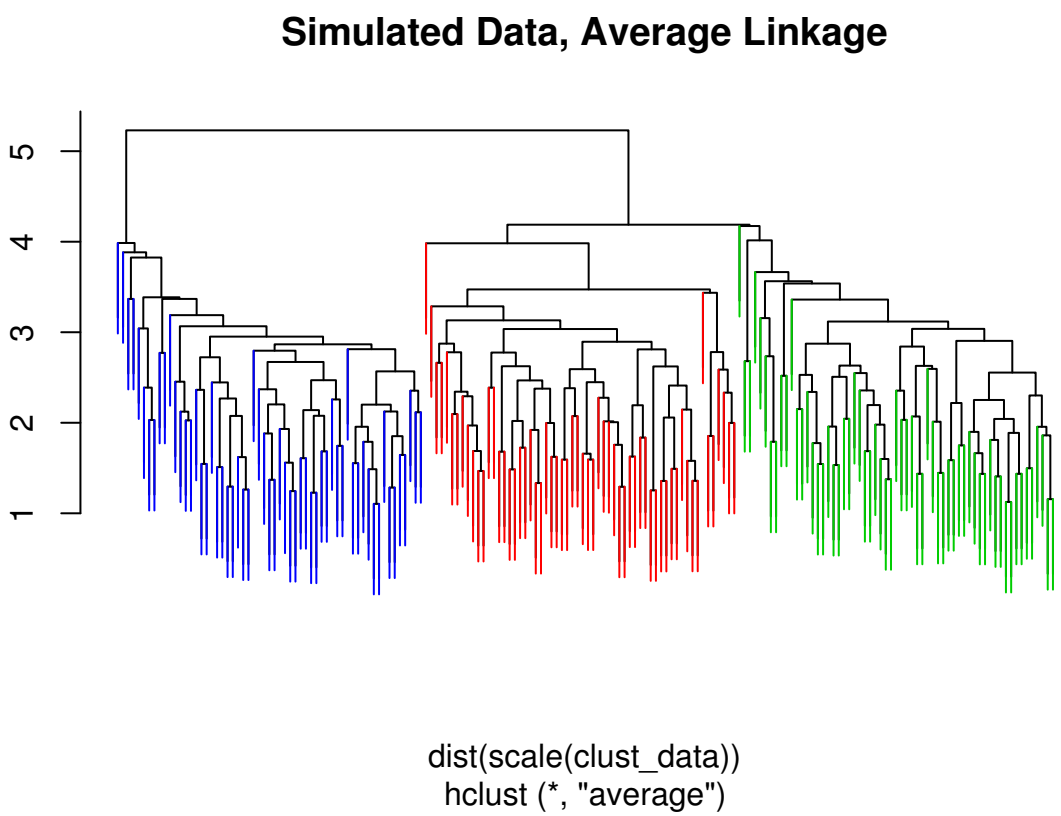
```
clust_data_hc = hclust(dist(scale(clust_data)), method = "single")
clust_data_cut = cutree(clust_data_hc , 3)
ColorDendrogram(clust_data_hc, y = clust_data_cut,
                labels = names(clust_data_cut),
                main = "Simulated Data, Single Linkage",
                branchlength = 0.5)
```




```
table(true_clusters, clust_data_cut)
```

```
##              clust_data_cut
## true_clusters  1  2  3
##             1 59  1  0
##             2 59  0  1
##             3 60  0  0
```

```
clust_data_hc = hclust(dist(scale(clust_data)), method = "average")
clust_data_cut = cutree(clust_data_hc , 3)
ColorDendrogram(clust_data_hc, y = clust_data_cut,
                labels = names(clust_data_cut),
                main = "Simulated Data, Average Linkage",
                branchlength = 1)
```




```
table(true_clusters, clust_data_cut)
```

```
##              clust_data_cut
## true_clusters  1  2  3
##             1  1 59  0
##             2  1  0 59
##             3 58  2  0
```

We also try single and average linkage. Single linkage seems to perform poorly here, while average linkage seems to be working well.

13.2.3 Iris Data

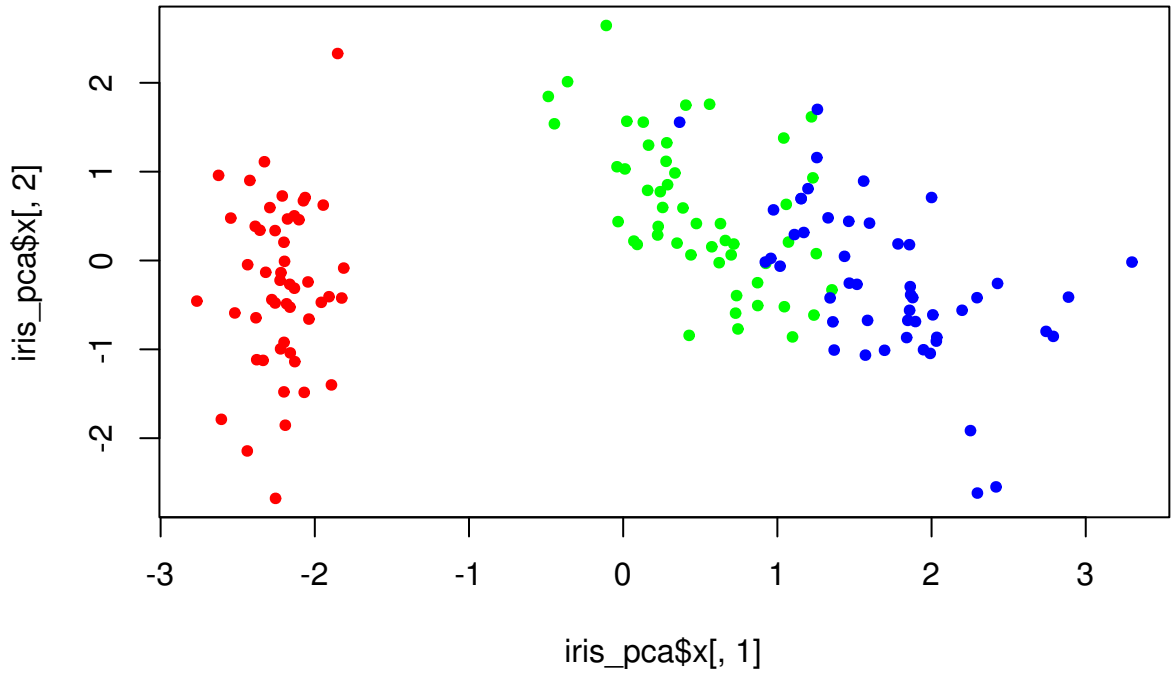
```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

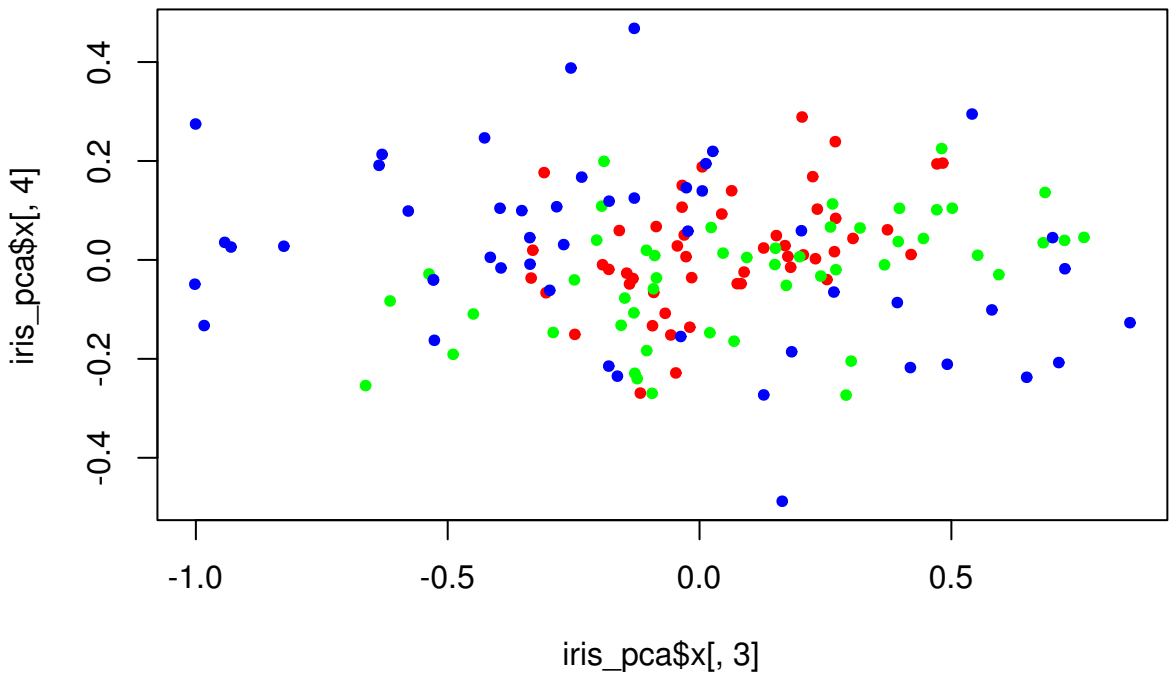
```
iris_pca = prcomp(iris[,-5], scale = TRUE)
iris_pca$rotation
```

```
##                     PC1         PC2        PC3        PC4
## Sepal.Length  0.5210659 -0.37741762  0.7195664  0.2612863
## Sepal.Width  -0.2693474 -0.92329566 -0.2443818 -0.1235096
## Petal.Length  0.5804131 -0.02449161 -0.1421264 -0.8014492
## Petal.Width   0.5648565 -0.06694199 -0.6342727  0.5235971
```

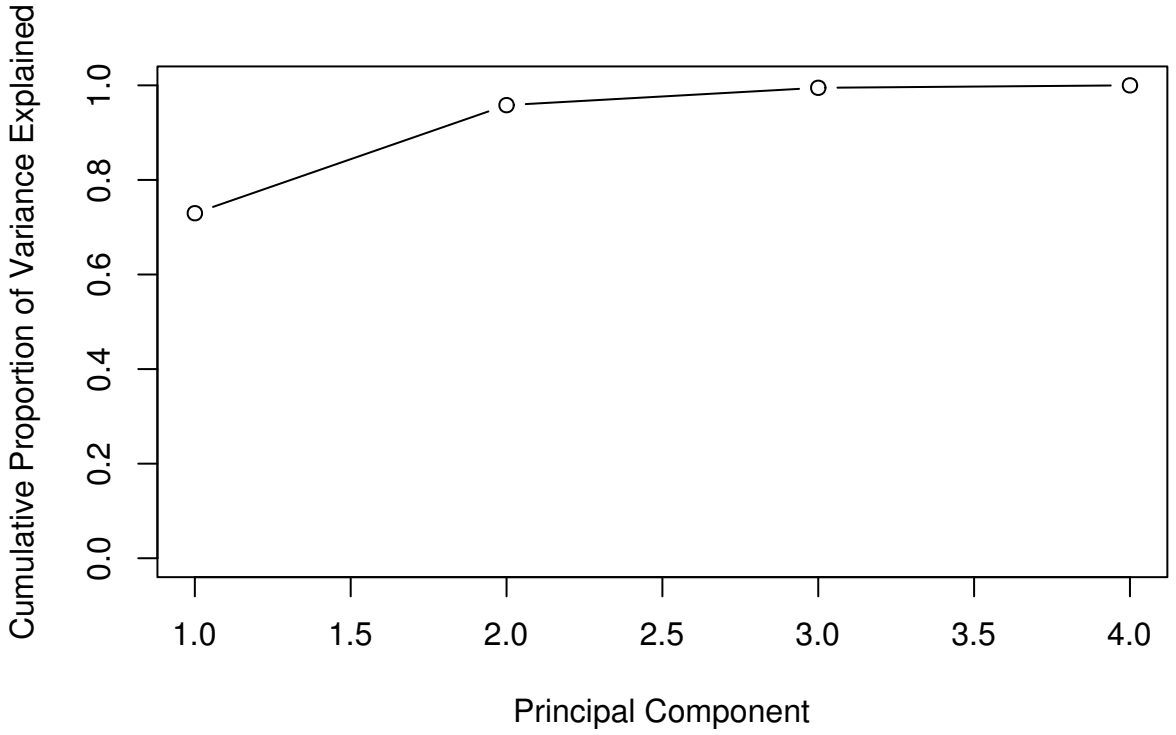
```
lab_to_col = function (labels){
  cols = rainbow (length(unique(labels)))
  cols[as.numeric (as.factor(labels))]
}

plot(iris_pca$x[,1], iris_pca$x[,2], col = lab_to_col(iris$Species), pch = 20)
```




```
plot(iris_pca$x[,3], iris_pca$x[,4], col = lab_to_col(iris$Species), pch = 20)
```



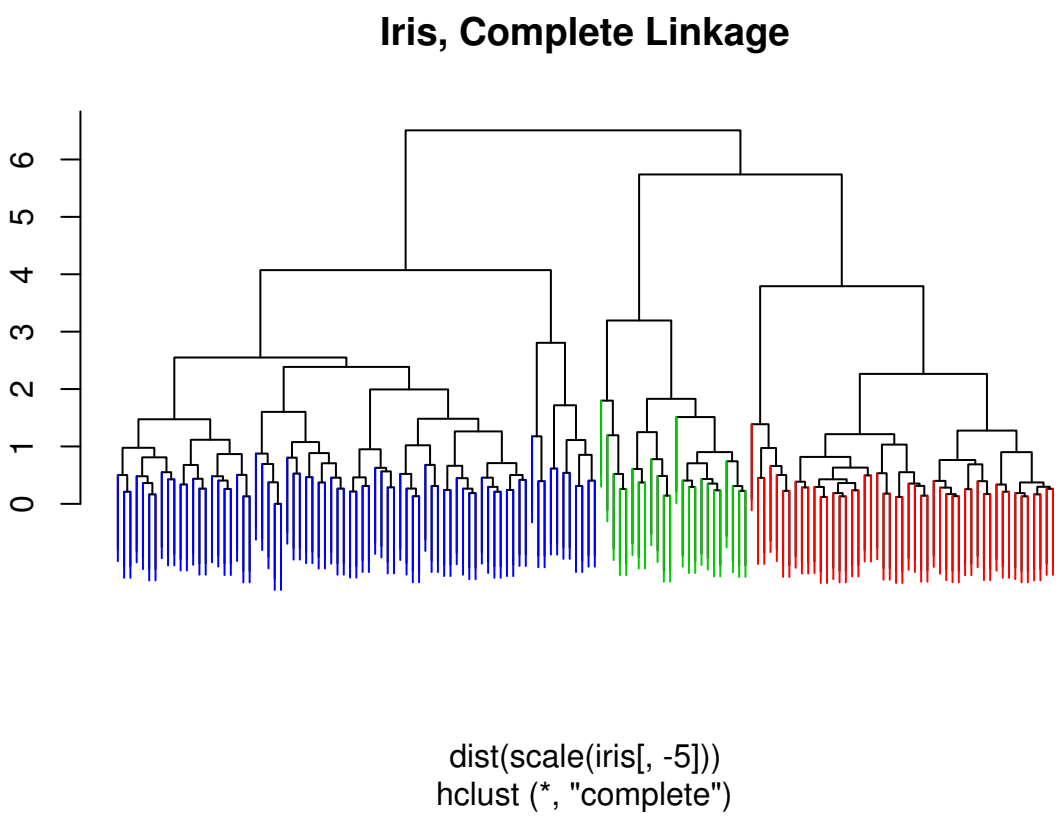

```
iris_pve = get_PVE(iris_pca)

plot(
  cumsum(iris_pve),
  xlab = "Principal Component",
  ylab = "Cumulative Proportion of Variance Explained",
  ylim = c(0, 1),
  type = 'b'
)
```




```
iris_kmeans = kmeans(iris[,-5], centers = 3, nstart = 10)
table(iris_kmeans$clust, iris[,5])
```

```
##
##     setosa versicolor virginica
##   1      0         48        14
##   2      0          2        36
##   3     50          0         0
```

```
iris_hc = hclust(dist(scale(iris[,-5])), method = "complete")
iris_cut = cutree(iris_hc , 3)
ColorDendrogram(iris_hc, y = iris_cut,
                labels = names(iris_cut),
                main = "Iris, Complete Linkage",
                branchlength = 1.5)
```



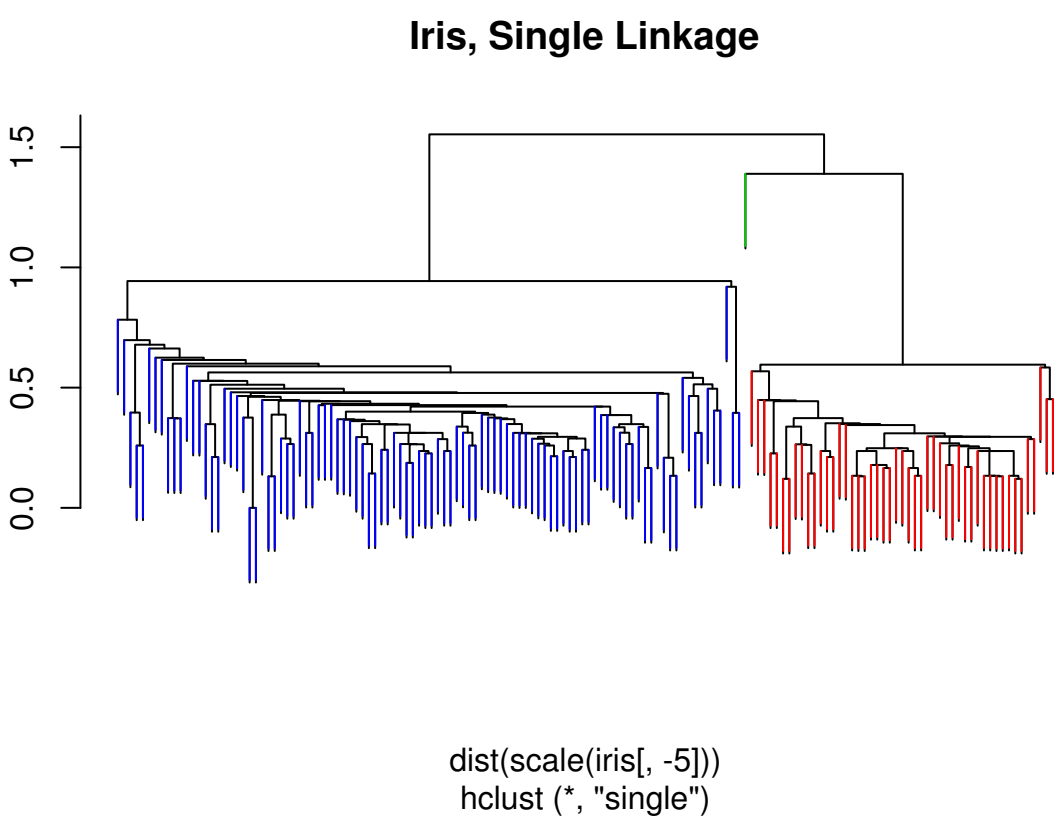

```
table(iris_cut, iris[,5])
```

```
##
## iris_cut setosa versicolor virginica
##        1     49          0         0
##        2      1         21         2
##        3      0         29        48
```

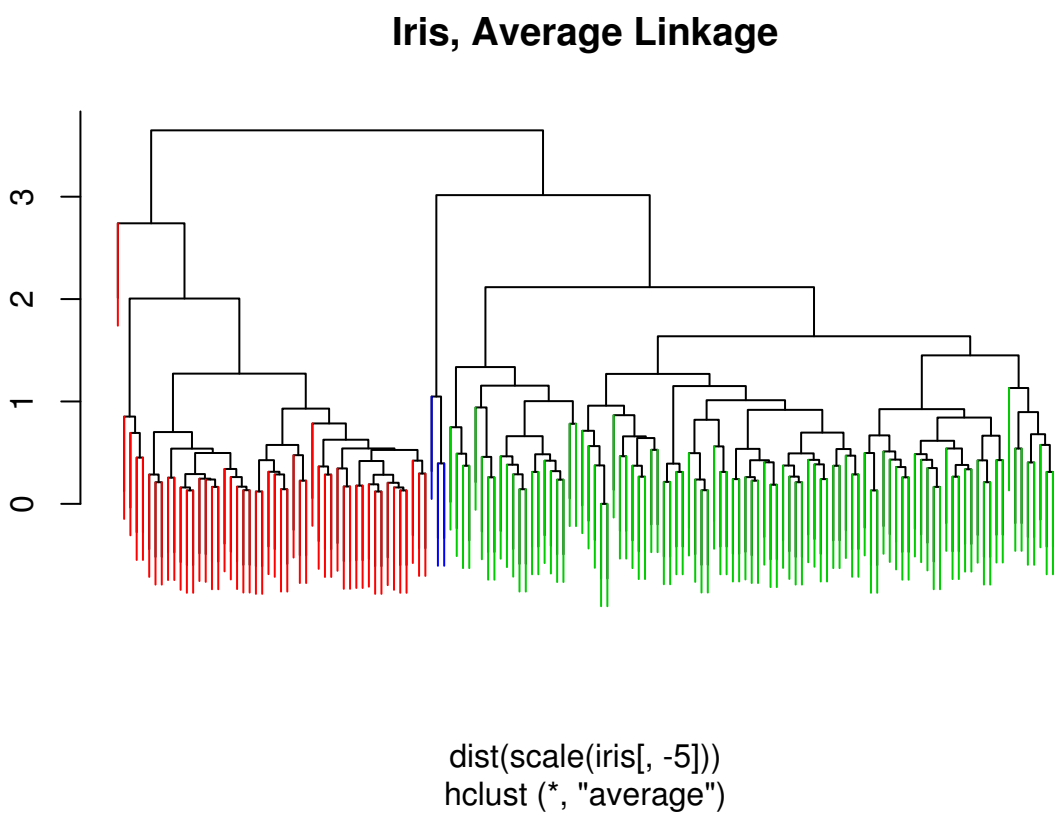
```
table(iris_cut, iris_kmeans$clust)
```

```
##
## iris_cut  1  2  3
##        1  0  0 49
##        2 23  0  1
##        3 39 38  0
```

```
iris_hc = hclust(dist(scale(iris[,-5])), method = "single")
iris_cut = cutree(iris_hc , 3)
ColorDendrogram(iris_hc, y = iris_cut,
                labels = names(iris_cut),
                main = "Iris, Single Linkage",
                branchlength = 0.3)
```




```
iris_hc = hclust(dist(scale(iris[,-5])), method = "average")
iris_cut = cutree(iris_hc , 3)
ColorDendrogram(iris_hc, y = iris_cut,
                labels = names(iris_cut),
                main = "Iris, Average Linkage",
                branchlength = 1)
```



13.3 External Links

- Hierarchical Cluster Analysis on Famous Data Sets - Using the `dendextend` package for in depth hierarchical cluster
- K-means Clustering is Not a Free Lunch - Comments on the assumptions made by K-means clustering.
- Principal Component Analysis - Explained Visually - Interactive PCA visualizations.

13.4 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "methods"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [7] "base"
```

- Additional Packages, Attached

```
## [1] "sparcl"  "MASS"    "mlbench" "caret"   "ggplot2" "lattice" "ISLR"
```

- Additional Packages, Not Attached

```
##  [1] "reshape2"     "kernlab"      "purrr"        "splines"
##  [5] "colorspace"   "stats4"       "htmltools"    "yaml"
##  [9] "survival"     "prodlim"      "rlang"        "e1071"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "foreach"      "plyr"         "bindr"        "dimRed"
## [21] "lava"         "robustbase"   "stringr"      "timeDate"
## [25] "munsell"      "gtable"       "recipes"      "codetools"
## [29] "evaluate"     "knitr"        "class"        "DEoptimR"
## [33] "Rcpp"         "scales"       "backports"    "ipred"
## [37] "CVST"         "digest"       "stringi"      "bookdown"
## [41] "dplyr"        "RcppRoll"     "ddalpha"      "grid"
## [45] "rprojroot"    "tools"        "magrittr"     "lazyeval"
## [49] "tibble"       "DRR"          "pkgconfig"    "Matrix"
## [53] "lubridate"    "gower"        "assertthat"   "rmarkdown"
## [57] "iterators"    "R6"           "rpart"        "nnet"
## [61] "nlme"         "compiler"
```

Chapter 14

Principal Component Analysis

-**TODO**: Add an example where `x1` an `x2` are very correlated. Compare regressions:

- `y ~ x1`
- `y ~ x2`
- `y ~ pc1` (almost as good)
- `y ~ x1 + x2` (true model form)

Chapter 15

k-Means

Chapter 16

Mixture Models

Chapter 17

Hierarchical Clustering

Part V

In Practice

Chapter 18

Overview

TODO: Discussion of necessary knowledge before reading book. Explain what will be recapped along the way.

Chapter 19

Supervised Learning Overview

At this point, you should know...

Bayes Classifier

- Classify to the class with the highest probability given a particular input x.

$$C^B(\mathbf{x}) = \underset{k}{\text{argmax}}\ P[Y = k \mid \mathbf{X} = \mathbf{x}]$$

- Since we rarely, if ever, know the true probabilities, use a classification method to estimate them using data.

The Bias-Variance Tradeoff

- As model complexity increases, **bias** decreases.
- As model complexity increases, **variance** increases.
- As a result, there is a model somewhere in the middle with the best accuracy. (Or lowest RMSE for regression.)

The Test-Train Split

- **Never use test data to train a model.** Test accuracy is a measure of how well a method works in general.
- We can identify underfitting and overfitting models relative to the best test accuracy.
  - A less complex model than the model with the best test accuracy is **underfitting**.
  - A more complex model than the model with the best test accuracy is **overfitting**.

Classification Methods

- Logistic Regression
- Linear Discriminant Analysis (LDA)
- Quadratic Discriminant Analysis (QDA)

- Naive Bayes (NB)
- k-Nearest Neighbors (KNN)
- For each, we can:
  - Obtain predicted probabilities.
  - Make classifications.
  - Find decision boundaries. (Seen only for some.)

Discriminative versus Generative Methods

- **Discriminative** methods learn the conditional distribution $p(y \mid x)$, thus could only simulate y given a fixed x.
- **Generative** methods learn the joint distribution $p(x, y)$, thus could only simulate new data (x, y).

Parametric and Non-Parametric Methods

- **Parametric** methods models $P[Y = k \mid X = x]$ as a specific function of parameters which are learned through data.
- **Non-Parametric** use an algorithmic approach to estimate $P[Y = k \mid X = x]$ for each possible input x.

Tuning Parameters

- Specify **how** to train a model. This in contrast to model parameters, which are learned through training.

Cross-Validation

- A method to estimate test metrics with training data. Repeats the train-validate split inside the training data.

Curse of Dimensionality

- As feature space grows, that is as p grows, "neighborhoods" must become much larger to contain "neighbors," thus local methods are not so local.

No-Free-Lunch Theorem

- There is no one classifier that will be best across all datasets.

19.1 External Links

- Wikipedia: No-Free-Lunch
- Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? - A paper that argues that No-Free-Lunch may be true in theory, but in practice there a only a few classifiers that outperform most others.

19.2 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1.

Chapter 20

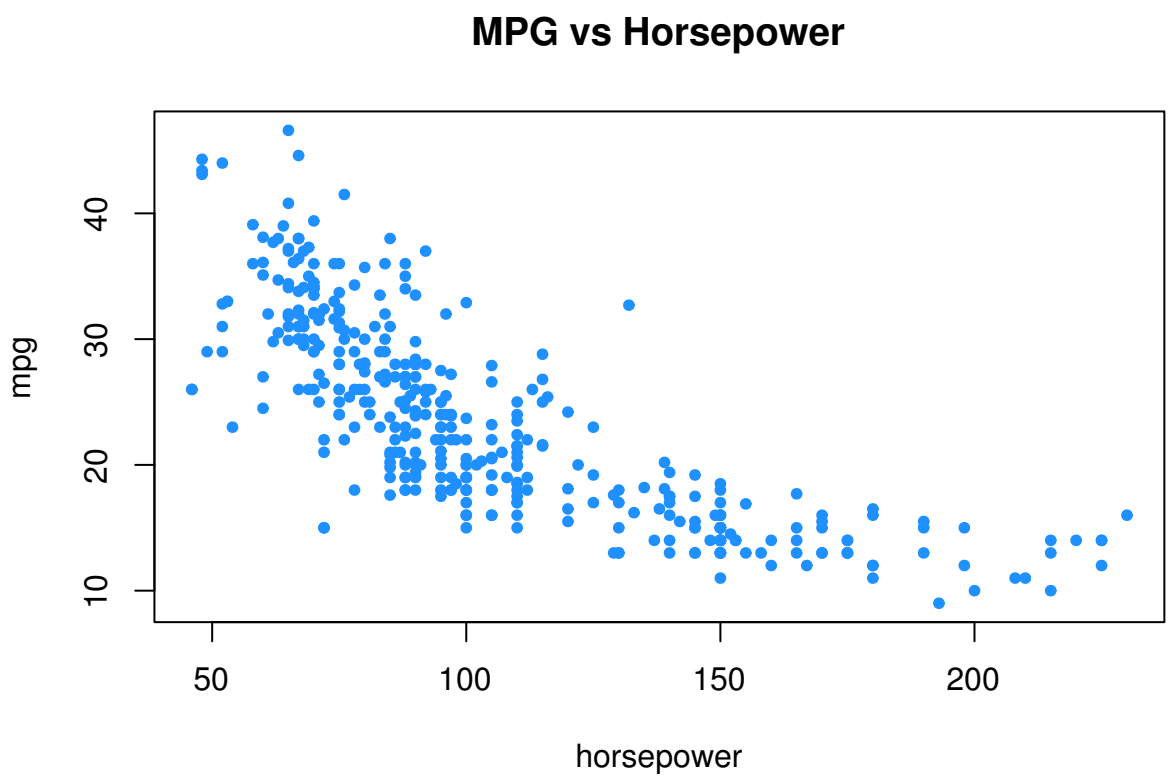
Resampling

In this chapter we introduce resampling methods including cross-validation and the bootstrap.

```
library(ISLR)
```

Here, we will use the `Auto` data from `ISLR` and attempt to predict `mpg` (a numeric variable) from `horsepower`.

```
## # A tibble: 392 x 9
##      mpg cylinders displacement horsepower weight acceleration  year
##  * <dbl>     <dbl>        <dbl>      <dbl>  <dbl>        <dbl> <dbl>
##  1    18         8          307        130   3504         12.0    70
##  2    15         8          350        165   3693         11.5    70
##  3    18         8          318        150   3436         11.0    70
##  4    16         8          304        150   3433         12.0    70
##  5    17         8          302        140   3449         10.5    70
##  6    15         8          429        198   4341         10.0    70
##  7    14         8          454        220   4354          9.0    70
##  8    14         8          440        215   4312          8.5    70
##  9    14         8          455        225   4425         10.0    70
## 10    15         8          390        190   3850          8.5    70
## # ... with 382 more rows, and 2 more variables: origin <dbl>, name <fctr>
```



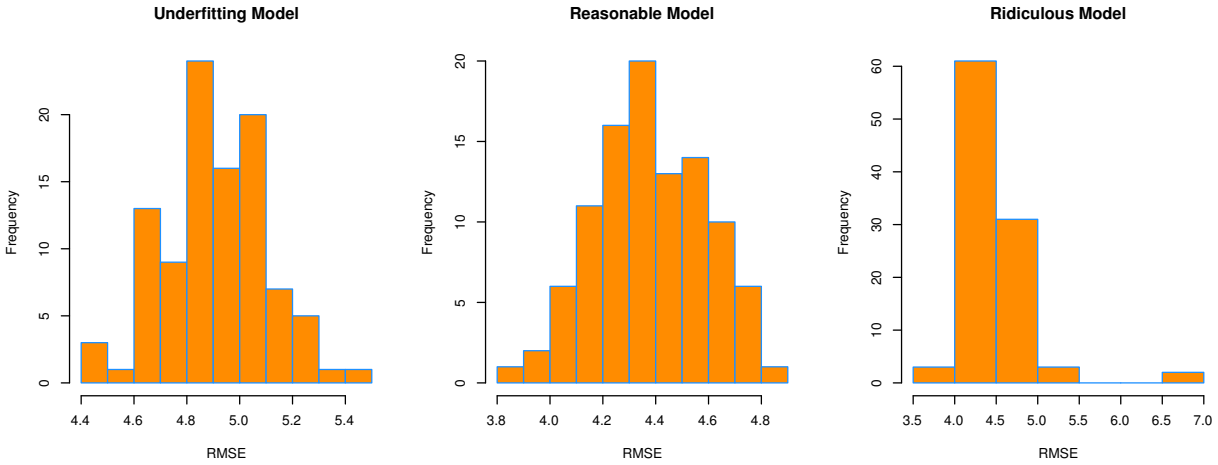
20.1 Test-Train Split

First, let's return to the usual test-train split procedure that we have used so far. Let's evaluate what happens if we repeat the process a large number of times, each time storing the test RMSE. We'll consider three models:

- An underfitting model: `mpg ~ horsepower`
- A reasonable model: `mpg ~ poly(horsepower, 2)`
- A ridiculous, overfitting model: `mpg ~ poly(horsepower, 8)`

```
set.seed(42)
num_reps = 100

lin_rmse  = rep(0, times = num_reps)
quad_rmse = rep(0, times = num_reps)
huge_rmse = rep(0, times = num_reps)

for(i in 1:100) {

  train_idx = sample(392, size = 196)

  lin_fit = lm(mpg ~ horsepower, data = Auto, subset = train_idx)
  lin_rmse[i] = sqrt(mean((Auto$mpg - predict(lin_fit, Auto))[-train_idx] ^ 2))
```

```
  quad_fit = lm(mpg ~ poly(horsepower, 2), data = Auto, subset = train_idx)
  quad_rmse[i] = sqrt(mean((Auto$mpg - predict(quad_fit, Auto))[-train_idx] ^ 2))

  huge_fit = lm(mpg ~ poly(horsepower, 8), data = Auto, subset = train_idx)
  huge_rmse[i] = sqrt(mean((Auto$mpg - predict(huge_fit, Auto))[-train_idx] ^ 2))
}
```



Notice two things, first that the "Reasonable" model has on average the smallest error. Second, notice large variability in the RMSE. We see this in the "Reasonable" model, but it is very clear in the "Ridiculous" model. Here it is very clear that if we use an "unlucky" split, our test error will be much larger than the likely reality.

20.2 Cross-Validation

Instead of using a single test-train split, we instead look to use cross-validation. There are many ways to perform cross-validation R, depending on the method of interest.

20.2.1 Method Specific

Some method, for example `glm()` through `cv.glm()` and `knn()` through `knn.cv()` have cross-validation capabilities built-in. We'll use `glm()` for illustration. First we need to convince ourselves that `glm()` can be used to perform the same tasks as `lm()`.

```
glm_fit = glm(mpg ~ horsepower, data = Auto)
coef(glm_fit)
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

```
lm_fit = lm(mpg ~ horsepower, data = Auto)
coef(lm_fit)
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

By default, `cv.glm()` will report leave-one-out cross-validation (LOOCV).

```
library(boot)
glm_fit = glm(mpg ~ horsepower, data = Auto)
loocv_rmse = sqrt(cv.glm(Auto, glm_fit)$delta)
loocv_rmse
```

```
## [1] 4.922552 4.922514
```

```
loocv_rmse[1]
```

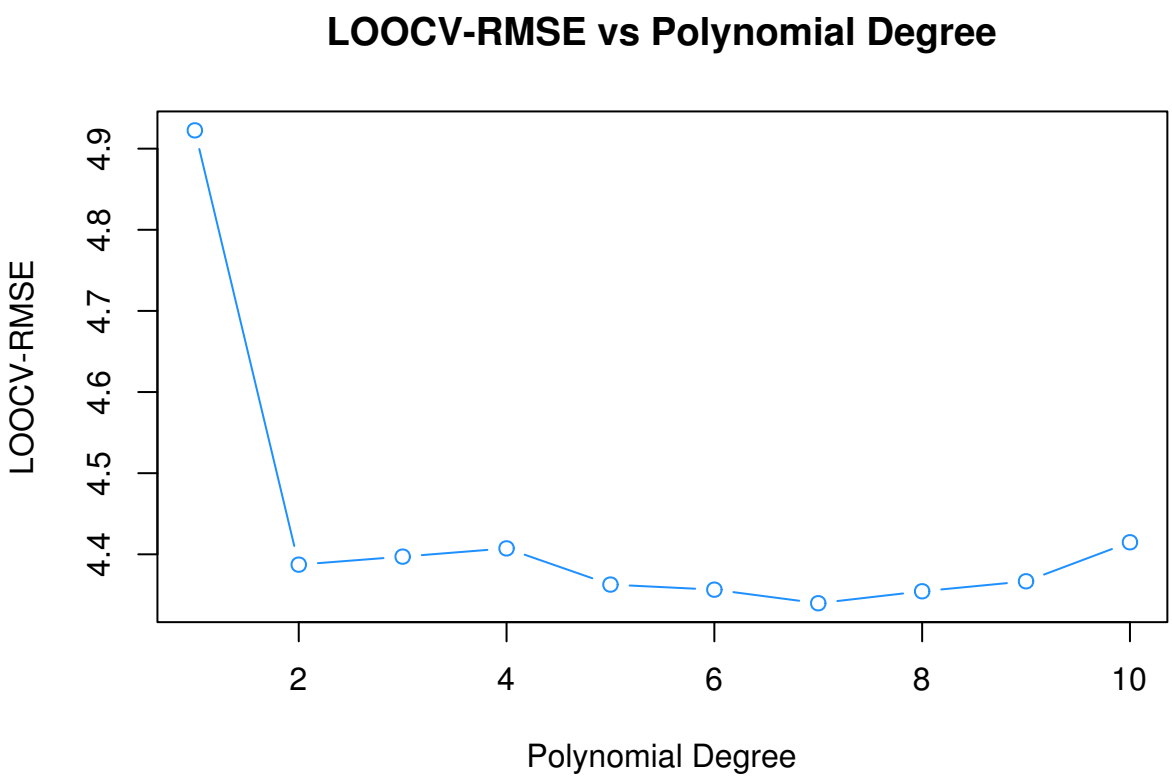
```
## [1] 4.922552
```

We are actually given two values. The first is exactly the LOOCV-RMSE. The second is a minor correct that we will not worry about. We take a square root to obtain LOOCV-RMSE.

```
loocv_rmse_poly = rep(0, times = 10)
for (i in seq_along(loocv_rmse_poly)) {
  glm_fit = glm(mpg ~ poly(horsepower, i), data = Auto)
  loocv_rmse_poly[i] = sqrt(cv.glm(Auto, glm_fit)$delta[1])
}
loocv_rmse_poly
```

```
##  [1] 4.922552 4.387279 4.397156 4.407316 4.362707 4.356449 4.339706
##  [8] 4.354440 4.366764 4.414854
```

```
plot(loocv_rmse_poly, type = "b", col = "dodgerblue",
     main = "LOOCV-RMSE vs Polynomial Degree",
     ylab = "LOOCV-RMSE", xlab = "Polynomial Degree")
```



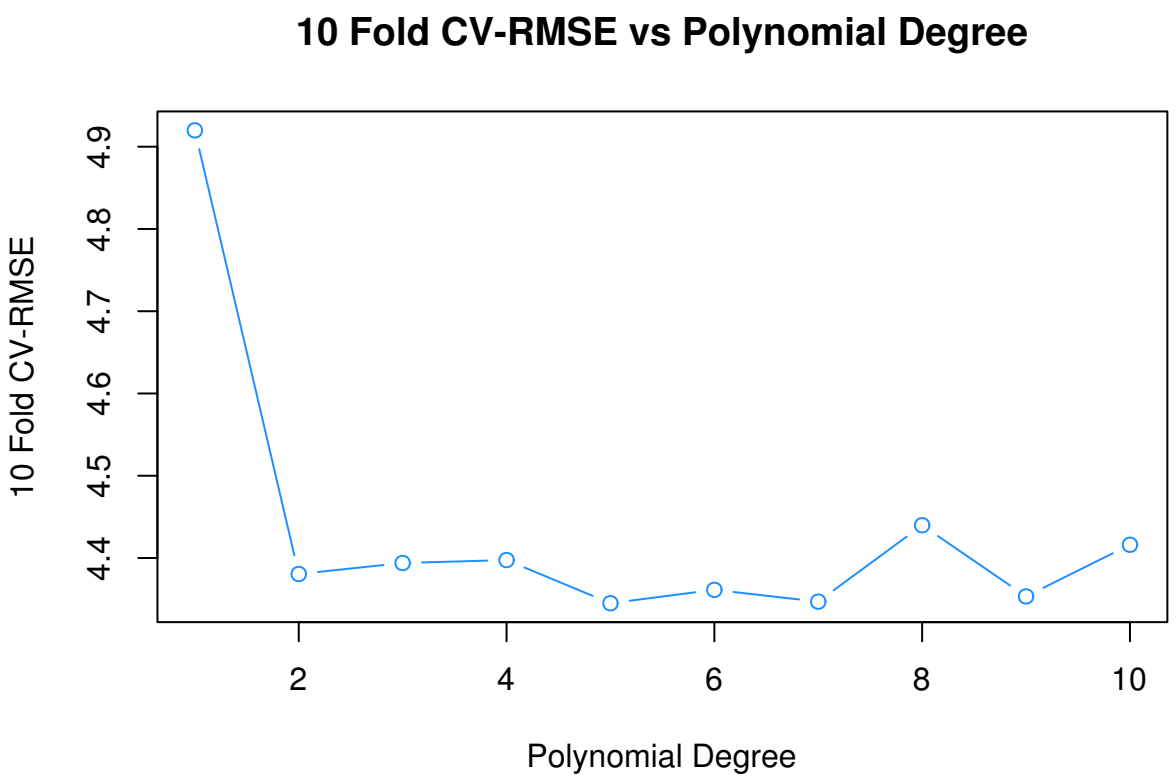
If you run the above code locally, you will notice that is painfully slow. We are fitting each of the 10 models 392 times, that is, each model n times, once with each data point left out. (Note: in this case, for a linear model, there is actually a shortcut formula which would allow us to obtain LOOCV-RMSE from a single fit to the data. See details in ISL as well as a link below.)

We could instead use k-fold cross-validation.

```
set.seed(17)
cv_10_rmse_poly = rep(0, times = 10)
for (i in seq_along(cv_10_rmse_poly)){
  glm_fit = glm(mpg ~ poly(horsepower, i), data = Auto)
  cv_10_rmse_poly[i] = sqrt(cv.glm(Auto, glm_fit, K = 10)$delta[1])
}
cv_10_rmse_poly
```

```
##  [1] 4.919878 4.380552 4.393929 4.397498 4.345010 4.361311 4.346963
##  [8] 4.439821 4.353321 4.416102
```

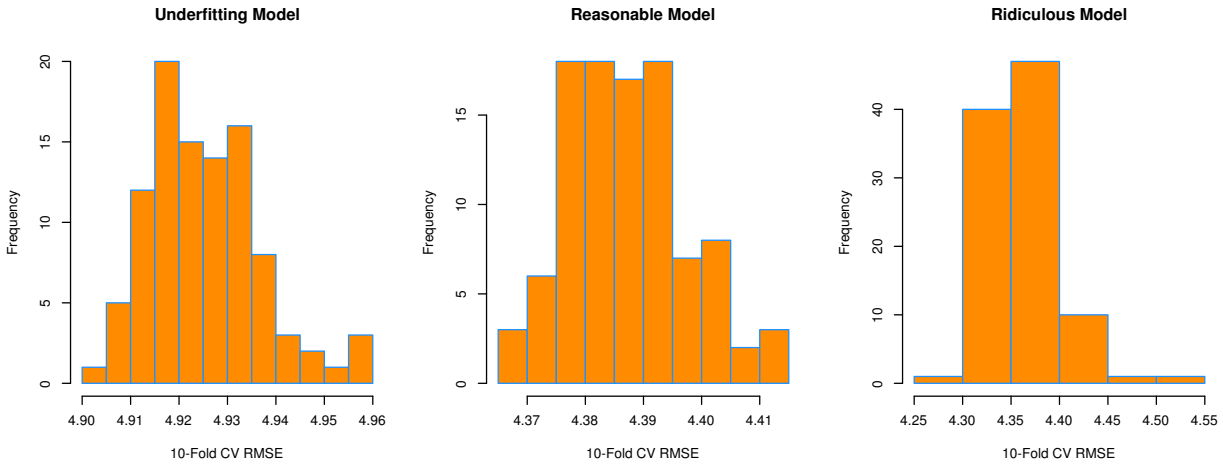
```
plot(cv_10_rmse_poly, type = "b", col = "dodgerblue",
     main = "10 Fold CV-RMSE vs Polynomial Degree",
     ylab = "10 Fold CV-RMSE", xlab = "Polynomial Degree")
```



Here we chose 10-fold cross-validation. Notice it is **much** faster. In practice, we usually stick to 5 or 10-fold CV.

```
set.seed(42)
num_reps = 100


lin_rmse_10_fold  = rep(0, times = num_reps)
quad_rmse_10_fold = rep(0, times = num_reps)
huge_rmse_10_fold = rep(0, times = num_reps)

for(i in 1:100) {

  lin_fit  = glm(mpg ~ poly(horsepower, 1), data = Auto)
  quad_fit = glm(mpg ~ poly(horsepower, 2), data = Auto)
  huge_fit = glm(mpg ~ poly(horsepower, 8), data = Auto)

  lin_rmse_10_fold[i]  = sqrt(cv.glm(Auto, lin_fit, K = 10)$delta[1])
  quad_rmse_10_fold[i] = sqrt(cv.glm(Auto, quad_fit, K = 10)$delta[1])
  huge_rmse_10_fold[i] = sqrt(cv.glm(Auto, huge_fit, K = 10)$delta[1])
}
```

Repeating the test-train split analysis from above, this time with 10-fold CV, see that that the resulting RMSE are much less variable. That means, will cross-validation still has some inherent randomness, it has a much smaller effect on the results.



20.2.2 Manual Cross-Validation

For methods that do not have a built-in ability to perform cross-validation, or for methods that have limited cross-validation capability, we will need to write our own code for cross-validation. (Spoiler: This is not true, but let's pretend it is, so we can see how to perform cross-validation from scratch.)

This essentially amounts to randomly splitting the data, then looping over the splits. The `createFolds()` function from the `caret()` package will make this much easier.

```
caret::createFolds(Auto$mpg)
```

```
## $Fold01
##  [1]  17  25  44  56  58  59  62  68  69  82  96  98 108 140 145 151 157
## [18] 160 163 174 181 190 194 200 214 216 240 242 278 282 288 321 323 330
## [35] 353 374 375 376 383
##
## $Fold02
##  [1]  21  22  33  46  47  64  70  81  85  95 121 130 134 148 156 158 161
## [18] 169 171 176 217 221 250 263 270 277 279 283 289 291 297 316 346 358
## [35] 371 377 380 384 386 392
##
## $Fold03
##  [1]  12  15  23  29  31  40  48  73  79  80  86  91  93 113 137 144 146
## [18] 177 183 188 199 201 203 206 208 225 231 243 247 251 265 267 273 296
## [35] 307 324 340 357 373
##
## $Fold04
##  [1]   3  18  34  42  50  51  52  54  71  76  87  88 103 106 107 164 170
## [18] 182 198 205 211 212 213 219 226 274 281 292 298 319 320 327 328 332
## [35] 337 360 364 381 385
##
## $Fold05
##  [1]   8  26  28  32  55  60  61  75  83  92  94  97 100 118 123 133 154
## [18] 159 172 184 209 220 222 236 237 241 244 253 272 310 322 335 341 349
## [35] 352 366 367 379 382
##
```

```
## $Fold06
##  [1]   7  13  30  43  63 101 102 109 114 135 153 155 168 178 180 191 192
## [18] 227 228 234 246 252 256 257 262 268 276 285 286 308 309 312 313 314
## [35] 318 326 331 334 363
##
## $Fold07
##  [1]   1   6  53  57  65  78  84  89  90 105 115 126 143 149 165 175 185
## [18] 207 224 229 230 233 258 260 264 271 290 293 300 305 343 344 345 348
## [35] 359 369 372 388 389 391
##
## $Fold08
##  [1]   2   4  11  16  41  45  49  66  77 104 110 119 120 122 124 136 138
## [18] 162 166 167 173 195 196 202 210 215 235 254 275 301 302 317 329 338
## [35] 342 350 362 368 378
##
## $Fold09
##  [1]  10  14  19  27  67  72  99 111 112 116 117 127 129 139 141 147 186
## [18] 187 193 204 218 223 238 248 249 261 294 295 303 306 315 325 336 339
## [35] 347 351 356 365
##
## $Fold10
##  [1]   5   9  20  24  35  36  37  38  39  74 125 128 131 132 142 150 152
## [18] 179 189 197 232 239 245 255 259 266 269 280 284 287 299 304 311 333
## [35] 354 355 361 370 387 390
```

Can you use this to verify the 10-fold CV results from above?

20.2.3 Test Data

The following example illustrates the need for a test set which is **never** used in model training. If for no other reason, it gives us a quick sanity check that we have cross-validated correctly.

To be specific we will test-train split the data, then perform cross-validation on the training data.

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

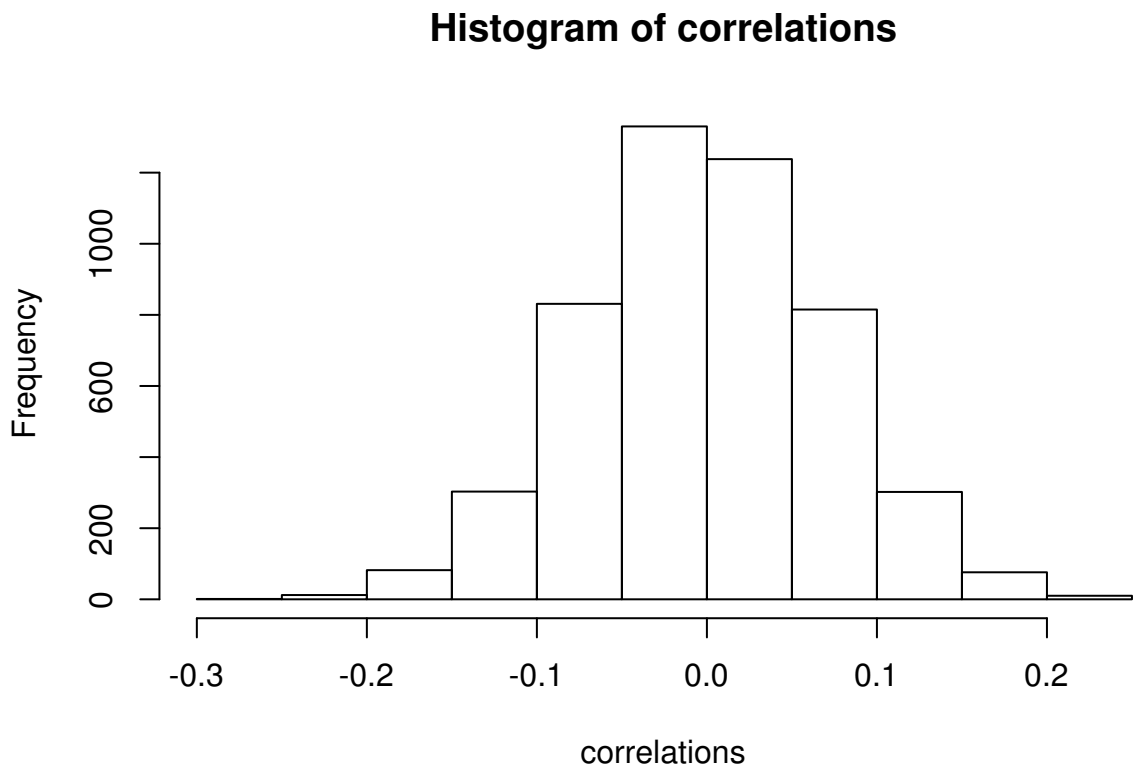
```
# simulate data
# y is 0/1
# X are independent N(0,1) variables
# X has no relationship with the response
# p >>> n
set.seed(430)
n = 400
p = 5000
X = replicate(p, rnorm(n))
y = c(rep(0, times = n / 4), rep(1, times = n / 4),
      rep(0, times = n / 4), rep(1, times = n / 4))
```

```
# first n/2 observations are used for training
# last n/2 observations used for testing
# both are 50% 0s and 50% 1s
```

```
# cv will be done inside train data
full_data = data.frame(y, X)
train = full_data[1:(n / 2), ]
test = full_data[((n / 2) + 1):n, ]
```

First, we use the screen-then-validate approach.

```
# find correlation between y and each predictor variable
correlations = apply(train[, -1], 2, cor, y = train$y)
hist(correlations)
```




```
# select the 25 largest (absolute) correlation
# these should be "useful" for prediction
selected = order(abs(correlations), decreasing = TRUE)[1:25]
correlations[selected]
```

```
##       X424      X4779      X2484      X1154      X2617      X1603
## -0.2577389  0.2491598  0.2379113 -0.2373367  0.2336055  0.2327971
##      X2963      X1091      X2806      X4586      X2569      X4532
##  0.2318932 -0.2281451 -0.2271382  0.2252979  0.2239974 -0.2225698
##      X3167       X741      X3329      X3862      X1741       X654
## -0.2201853 -0.2188919 -0.2186248 -0.2174146 -0.2150666  0.2130732
##      X3786      X4617      X3296      X2295       X999      X4349
##  0.2090650 -0.2086551 -0.2075271 -0.2072127  0.2055167 -0.1995252
```

```
##      X1409
##  0.1977006
```

```
# subset the test and training data based on the selected predictors
train_screen = train[c(1, selected)]
test_screen = test[c(1, selected)]

# fit an additive logistic regression
# use 10-fold cross-validation to obtain an estimate of test accuracy
# horribly optimistic
library(boot)
glm_fit = glm(y ~ ., data = train_screen, family = "binomial")
1 - cv.glm(train_screen, glm_fit, K = 10)$delta[1]
```

```
## [1] 0.709234
```

```
# get test accuracy, which we expect to be 0.50
# no better than guessing
glm_pred = (predict(glm_fit, newdata = test_screen, type = "response") > 0.5) * 1
accuracy(predicted = glm_pred, actual = test_screen$y)
```

```
## [1] 0.46
```

Now, we will correctly screen-while-validating.

```
# use the caret package to obtain 10 "folds"
folds = caret::createFolds(train_screen$y)

# for each fold
# - pre-screen variables on the 9 training folds
# - fit model to these variables
# - get accuracy on validation fold
fold_acc = rep(0, length(folds))

for(i in seq_along(folds)) {

  # split for fold i
  train_fold = train[-folds[[i]],]
  validate_fold = train[folds[[i]],]

  # screening for fold i
  correlations = apply(train_fold[, -1], 2, cor, y = train_fold[,1])
  selected = order(abs(correlations), decreasing = TRUE)[1:25]
  train_fold_screen = train_fold[ ,c(1,selected)]
  validate_fold_screen = validate_fold[ ,c(1,selected)]

  # accuracy for fold i
  glm_fit = glm(y ~ ., data = train_fold_screen, family = "binomial")
  glm_pred = (predict(glm_fit, newdata = validate_fold_screen, type = "response") > 0.5)*1
  fold_acc[i] = mean(glm_pred == validate_fold_screen$y)

}
```

```
# report all 10 validation fold accuracies
fold_acc
```

```
##  [1] 0.45 0.40 0.50 0.35 0.50 0.35 0.45 0.50 0.60 0.50
```

```
# properly cross-validated error
# this roughly matches what we expect in the test set
mean(fold_acc)
```

```
## [1] 0.46
```

20.3 Bootstrap

ISL also discusses the bootstrap, which is another resampling method. However, it is less relevant to the statistical learning tasks we will encounter. It could be useful if we were to attempt to calculate the bias and variance of a prediction (estimate) without access to the data generating process. Return to the bias-variance tradeoff chapter and think about how the bootstrap could be used to obtain estimates of bias and variance with a single dataset, instead of repeated simulated datasets.

For fun, write-up a simulation study which compares the strategy in the bias-variance tradeoff chapter to a strategy using bootstrap resampling of a single dataset. Submit it to be added to this chapter!

20.4 External Links

- YouTube: Cross-Validation, Part 1 - Video from user "mathematicalmonk" which introduces K-fold cross-validation in greater detail.
  - YouTube: Cross-Validation, Part 2 - Continuation which discusses selection and resampling strategies.
  - YouTube: Cross-Validation, Part 3 - Continuation which discusses choice of K.
- Blog: Fast Computation of Cross-Validation in Linear Models - Details for using leverage to speed-up LOOCV for linear models.
- OTexts: Bootstrap - Some brief mathematical details of the bootstrap.

20.5 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
## [1] "boot" "ISLR"
```

- Additional Packages, Not Attached

```
##  [1] "purrr"        "reshape2"     "kernlab"      "splines"
##  [5] "lattice"      "colorspace"   "stats4"       "htmltools"
##  [9] "yaml"         "survival"     "prodlim"      "rlang"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "foreach"      "plyr"         "bindr"        "dimRed"
## [21] "lava"         "robustbase"   "stringr"      "timeDate"
## [25] "munsell"      "gtable"       "recipes"      "codetools"
## [29] "evaluate"     "knitr"        "caret"        "class"
## [33] "DEoptimR"     "methods"      "Rcpp"         "scales"
## [37] "backports"    "ipred"        "CVST"         "ggplot2"
## [41] "digest"       "stringi"      "bookdown"     "dplyr"
## [45] "RcppRoll"     "ddalpha"      "grid"         "rprojroot"
## [49] "tools"        "magrittr"     "lazyeval"     "tibble"
## [53] "DRR"          "pkgconfig"    "MASS"         "Matrix"
## [57] "lubridate"    "gower"        "assertthat"   "rmarkdown"
## [61] "iterators"    "R6"           "rpart"        "nnet"
## [65] "nlme"         "compiler"
```

Chapter 21

The `caret` Package

Instructor's Note: This chapter is currently missing the usual narrative text. Hopefully it will be added later.

Now that we have seen a number of classification (and regression) methods, and introduced cross-validation, we see the general outline of a predictive analysis:

- Select a method
- Test-train split the available data
- Decide on a set of candidate models via tuning parameters
- Select the best model (tuning parameters) using a cross-validated metric
- Use chosen model to make predictions
- Calculate relevant metrics on the test data

At face value it would seem like it should be easy to repeat this process for a number of different methods, however we have run into a number of difficulties attempting to do so with `R`.

- The `predict()` function seems to have a different behavior for each new method we see.
- Many methods have different cross-validation functions, or worse yet, no built-in process for cross-validation.
- Not all methods expect the same data format. Some methods do not use formula syntax.
- Different methods have different handling of categorical predictors.

Thankfully, the `R` community has essentially provided a silver bullet for these issues, the `caret` package. Returning to the above list, we will see that a number of these tasks are directly addressed in the `caret` package.

- Test-train split the available data
  - `createDataPartition()` will take the place of our manual data splitting. It will also do some extra work to ensure that the train and test samples are somewhat similar.
- Decide on a set of candidate models via tuning parameters
  - `expand.grid()` is not a function in `caret`, but we will get in the habit of using it to specify a grid of tuning parameters.
- Select the best model (tuning parameters) using a cross-validated metric
  - `trainControl()` will setup cross-validation

  - `train()` is the workhorse of `caret`. It takes the following information then trains the requested model:
    * `form`, a formula, such as `y ~ .`
    * `data`
    * `method`, from a long list of possibilities
    * `preProcess` which allows for specification of things such as centering and scaling
    * `tuneGrid` which specifies the tuning parameters to train over
    * `trControl` which specifies the resampling scheme, that is, how cross-validation should be performed

- Use chosen model to make predictions
  - `predict()` used on objects of type `train` will be magical!

To illustrate `caret`, we return to our familiar `Default` data.

```
data(Default, package = "ISLR")
```

```
library(caret)
```

We first test-train split the data using `createDataPartition`. Here we are using 75% of the data for training.

```
set.seed(430)
default_idx = createDataPartition(Default$default, p = 0.75, list = FALSE)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

```
default_glm = train(
  form = default ~ .,
  data = default_trn,
  method = "glm",
  family = "binomial",
  trControl = trainControl(method = "cv", number = 5)
)
```

```
default_glm
```

```
## Generalized Linear Model
##
## 7501 samples
##    3 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 6001, 6001, 6000, 6001, 6001
## Resampling results:
##
##   Accuracy   Kappa
##   0.9733372  0.4174282
```

```
names(default_glm)
```

```
##  [1] "method"       "modelInfo"    "modelType"    "results"
##  [5] "pred"         "bestTune"     "call"         "dots"
##  [9] "metric"       "control"      "finalModel"   "preProcess"
## [13] "trainingData" "resample"     "resampledCM"  "perfNames"
## [17] "maximize"     "yLimits"      "times"        "levels"
## [21] "terms"        "coefnames"    "contrasts"    "xlevels"
```

```
default_glm$results
```

```
##   parameter  Accuracy     Kappa AccuracySD   KappaSD
## 1      none 0.9733372 0.4174282 0.00358649 0.1180854
```

```
default_glm$finalModel
```

```
##
## Call:  NULL
##
## Coefficients:
## (Intercept)   studentYes      balance       income
##  -1.066e+01   -6.254e-01    5.647e-03    1.395e-06
##
## Degrees of Freedom: 7500 Total (i.e. Null);  7497 Residual
## Null Deviance:       2192
## Residual Deviance: 1204  AIC: 1212
```

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

```
# make predictions
head(predict(default_glm, newdata = default_trn))
```

```
## [1] No No No No No No
## Levels: No Yes
```

```
# train acc
accuracy(actual = default_trn$default,
         predicted = predict(default_glm, newdata = default_trn))
```

```
## [1] 0.9730703
```

```
# test acc
accuracy(actual = default_tst$default,
         predicted = predict(default_glm, newdata = default_tst))
```

```
## [1] 0.9739896
```

```
# get probs
head(predict(default_glm, newdata = default_trn, type = "prob"))
```

```
##          No         Yes
## 1 0.9984674 0.001532637
## 3 0.9895850 0.010414985
## 5 0.9979141 0.002085863
## 6 0.9977233 0.002276746
## 8 0.9987645 0.001235527
## 9 0.9829081 0.017091877
```

```
default_knn = train(
  default ~ .,
  data = default_trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5)
)
```

```
default_knn
```

```
## k-Nearest Neighbors
##
## 7501 samples
##    3 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 6001, 6000, 6001, 6001, 6001
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.9660044  0.14910366
##   7  0.9654711  0.08890944
##   9  0.9660044  0.03400684
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 9.
```

```
default_knn = train(
  default ~ .,
  data = default_trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  preProcess = c("center", "scale"),
  tuneGrid = expand.grid(k = seq(1, 100, by = 1))
)
```

```
default_knn
```

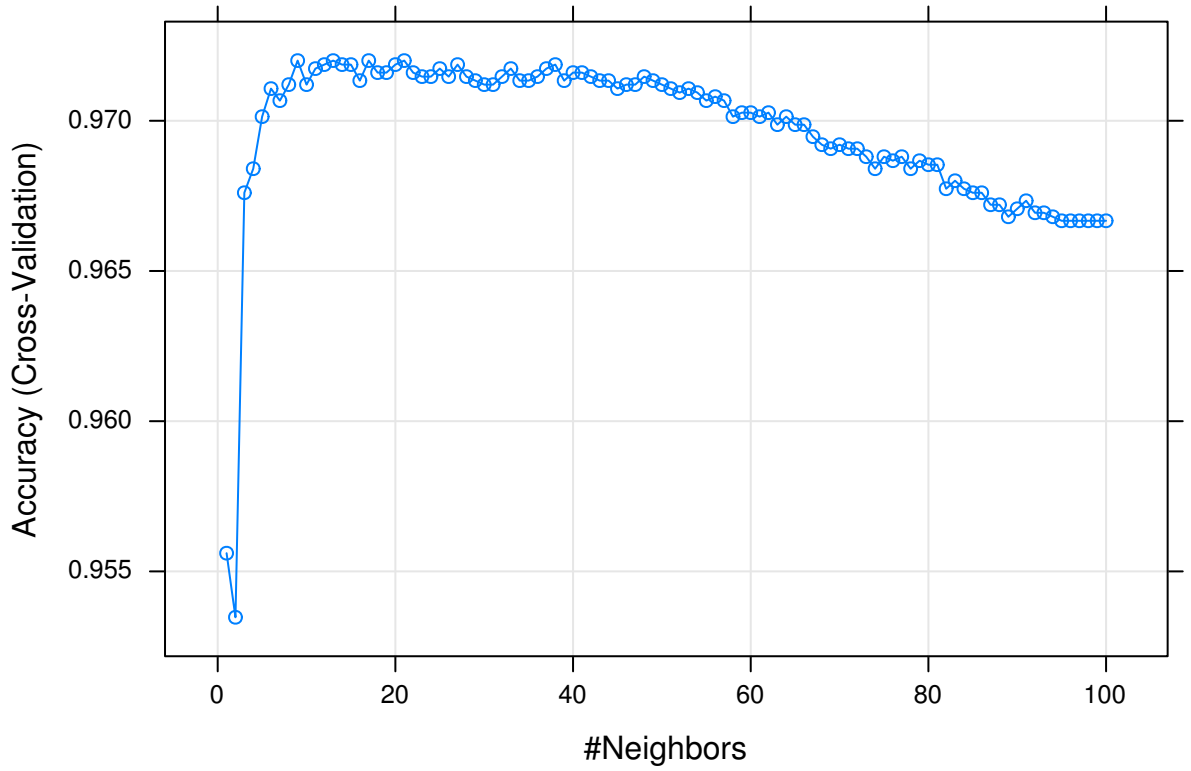
```
## k-Nearest Neighbors
```

```
##
## 7501 samples
##    3 predictor
##    2 classes: 'No', 'Yes'
##
## Pre-processing: centered (3), scaled (3)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 6001, 6000, 6001, 6001, 6001
## Resampling results across tuning parameters:
##
##   k    Accuracy   Kappa
##     1  0.9556062  0.298089557
##     2  0.9534727  0.278858080
##     3  0.9676048  0.375302287
##     4  0.9684043  0.392531223
##     5  0.9701374  0.404022317
##     6  0.9710706  0.423064304
##     7  0.9706704  0.390664155
##     8  0.9712037  0.399173401
##     9  0.9720034  0.409106426
##    10  0.9712037  0.386139169
##    11  0.9717367  0.391087258
##    12  0.9718700  0.382391562
##    13  0.9720034  0.383930148
##    14  0.9718700  0.385920266
##    15  0.9718701  0.379873523
##    16  0.9713369  0.377628107
##    17  0.9720036  0.380320495
##    18  0.9716036  0.373394281
##    19  0.9716036  0.365929261
##    20  0.9718700  0.368421254
##    21  0.9720035  0.364668426
##    22  0.9716036  0.362097470
##    23  0.9714701  0.356376683
##    24  0.9714701  0.355664780
##    25  0.9717365  0.350899581
##    26  0.9714701  0.345911449
##    27  0.9718700  0.353134003
##    28  0.9714703  0.345081109
##    29  0.9713370  0.343455277
##    30  0.9712036  0.339093085
##    31  0.9712037  0.339182082
##    32  0.9714703  0.342168206
##    33  0.9717369  0.352226337
##    34  0.9713370  0.341045339
##    35  0.9713370  0.334327819
##    36  0.9714703  0.339159698
##    37  0.9717369  0.338120724
##    38  0.9718701  0.338963642
##    39  0.9713369  0.326523756
##    40  0.9716036  0.328850300
##    41  0.9716035  0.325286501
##    42  0.9714701  0.319449330
##    43  0.9713370  0.311058773
```

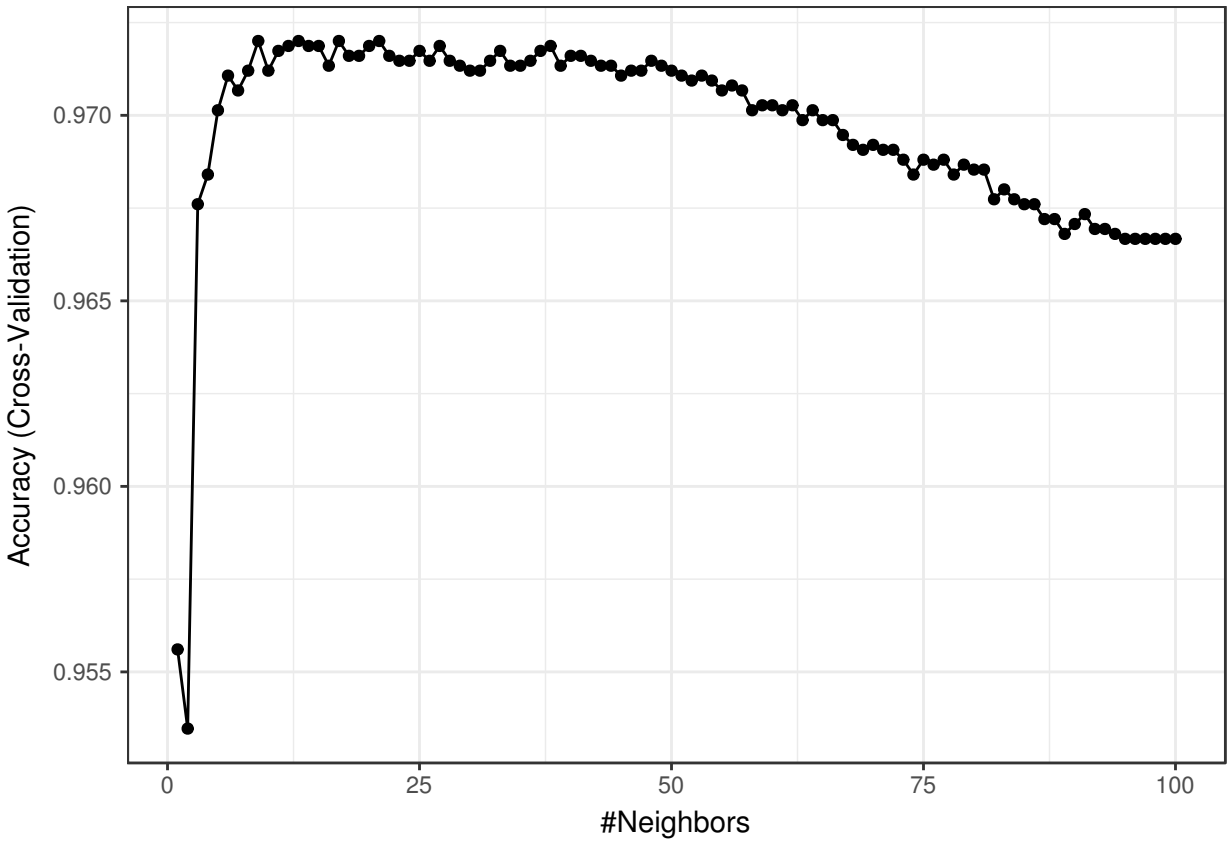
```
##     44  0.9713369  0.314958826
##     45  0.9710704  0.304969687
##     46  0.9712037  0.310515120
##     47  0.9712036  0.305970685
##     48  0.9714702  0.307416051
##     49  0.9713370  0.303143005
##     50  0.9712036  0.297926950
##     51  0.9710704  0.287872960
##     52  0.9709372  0.282257433
##     53  0.9710705  0.283589156
##     54  0.9709371  0.282226306
##     55  0.9706705  0.266533647
##     56  0.9708038  0.271977860
##     57  0.9706706  0.261835380
##     58  0.9701372  0.249188016
##     59  0.9702706  0.250148772
##     60  0.9702706  0.250114697
##     61  0.9701372  0.239629655
##     62  0.9702708  0.240658770
##     63  0.9698707  0.222966445
##     64  0.9701372  0.234377021
##     65  0.9698705  0.217685465
##     66  0.9698706  0.213155702
##     67  0.9694707  0.199923192
##     68  0.9692041  0.187559137
##     69  0.9690708  0.181218961
##     70  0.9692041  0.182387968
##     71  0.9690708  0.176074591
##     72  0.9690709  0.170120508
##     73  0.9688042  0.163573197
##     74  0.9684042  0.150065236
##     75  0.9688042  0.152290768
##     76  0.9686708  0.151535913
##     77  0.9688042  0.151657123
##     78  0.9684043  0.138012553
##     79  0.9686709  0.139859409
##     80  0.9685375  0.127092858
##     81  0.9685375  0.121296450
##     82  0.9677376  0.085964305
##     83  0.9680043  0.093510882
##     84  0.9677376  0.085481828
##     85  0.9676043  0.072888526
##     86  0.9676044  0.077956439
##     87  0.9672044  0.050919251
##     88  0.9672044  0.043952211
##     89  0.9668044  0.028789826
##     90  0.9670711  0.029972813
##     91  0.9673377  0.044609786
##     92  0.9669378  0.022381190
##     93  0.9669377  0.022513257
##     94  0.9668044  0.014921466
##     95  0.9666711  0.007329843
##     96  0.9666711  0.007329843
##     97  0.9666711  0.007329843
```

```
##    98  0.9666711  0.007329843
##    99  0.9666711  0.007329843
##   100  0.9666711  0.007329843
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 17.
```

```
plot(default_knn)
```




```
ggplot(default_knn) + theme_bw()
```




```
default_knn$bestTune
```

```
##     k
## 17 17
```

```
get_best_result = function(caret_fit) {
  best_result = caret_fit$results[as.numeric(rownames(caret_fit$bestTune)), ]
  rownames(best_result) = NULL
  best_result
}
```

```
get_best_result(default_knn)
```

```
##    k  Accuracy     Kappa  AccuracySD    KappaSD
## 1 17 0.9720036 0.3803205 0.002404977 0.05972573
```

```
default_knn$finalModel
```

```
## 17-nearest neighbor classification model
## Training set class distribution:
##
##   No  Yes
## 7251  250
```

Notes to add later:

- Fewer ties with CV than simple test-train approach
- Default grid vs specified grid. `tuneLength`
- Create table summarizing results for `knn()` and `glm()`. Test, train, and CV accuracy. Maybe also show SD for CV.

21.1 External Links

- The `caret` Package - Reference documentation for the `caret` package in `bookdown` format.
- `caret` Model List - List of available models in `caret`.

21.2 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "methods"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [7] "base"
```

- Additional Packages, Attached

```
## [1] "caret"   "ggplot2" "lattice"
```

- Additional Packages, Not Attached

```
##  [1] "reshape2"     "kernlab"      "purrr"        "splines"
##  [5] "colorspace"   "stats4"       "htmltools"    "yaml"
##  [9] "survival"     "prodlim"      "rlang"        "e1071"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "foreach"      "plyr"         "bindr"        "dimRed"
## [21] "lava"         "robustbase"   "stringr"      "timeDate"
## [25] "munsell"      "gtable"       "recipes"      "codetools"
## [29] "evaluate"     "labeling"     "knitr"        "class"
## [33] "DEoptimR"     "Rcpp"         "scales"       "backports"
## [37] "ipred"        "CVST"         "digest"       "stringi"
## [41] "bookdown"     "dplyr"        "RcppRoll"     "ddalpha"
## [45] "grid"         "rprojroot"    "tools"        "magrittr"
## [49] "lazyeval"     "tibble"       "DRR"          "pkgconfig"
## [53] "MASS"         "Matrix"       "lubridate"    "gower"
## [57] "assertthat"   "rmarkdown"    "iterators"    "R6"
## [61] "rpart"        "nnet"         "nlme"         "compiler"
```

Chapter 22

Subset Selection

Instructor's Note: This chapter is currently missing the usual narrative text. Hopefully it will be added later.

```
data(Hitters, package = "ISLR")
```

```
sum(is.na(Hitters))
```

```
## [1] 59
```

```
sum(is.na(Hitters$Salary))
```

```
## [1] 59
```

```
Hitters = na.omit(Hitters)
sum(is.na(Hitters))
```

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

22.1 AIC, BIC, and Cp

22.1.1 `leaps` Package

```
library(leaps)
```

22.1.2 Best Subset

```
fit_all = regsubsets(Salary ~ ., Hitters)
summary(fit_all)
```

```
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., Hitters)
## 19 Variables  (and intercept)
##            Forced in Forced out
## AtBat          FALSE      FALSE
## Hits           FALSE      FALSE
## HmRun          FALSE      FALSE
## Runs           FALSE      FALSE
## RBI            FALSE      FALSE
## Walks          FALSE      FALSE
## Years          FALSE      FALSE
## CAtBat         FALSE      FALSE
## CHits          FALSE      FALSE
## CHmRun         FALSE      FALSE
## CRuns          FALSE      FALSE
## CRBI           FALSE      FALSE
## CWalks         FALSE      FALSE
## LeagueN        FALSE      FALSE
## DivisionW      FALSE      FALSE
## PutOuts        FALSE      FALSE
## Assists        FALSE      FALSE
## Errors         FALSE      FALSE
## NewLeagueN     FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##          AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns
## 1  ( 1 ) " "   " "  " "   " "  " " " " " "   " "   " "   " "    " "    " "
## 2  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "   " "    " "    " "
## 3  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "   " "    " "    " "
## 4  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "   " "    " "    " "
## 5  ( 1 ) "*"   "*"  " "   " "  " " " " " "   " "   " "   " "    " "    " "
## 6  ( 1 ) "*"   "*"  " "   " "  " " " " "*"   " "   " "   " "    " "    " "
## 7  ( 1 ) " "   "*"  " "   " "  " " " " "*"   " "   "*"   "*"    "*"    " "
## 8  ( 1 ) "*"   "*"  " "   " "  " " " " "*"   " "   " "   " "    "*"    "*"
##          CRBI CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1  ( 1 ) "*"  " "    " "     " "       " "     " "     " "    " "
## 2  ( 1 ) "*"  " "    " "     " "       " "     " "     " "    " "
## 3  ( 1 ) "*"  " "    " "     " "       "*"     " "     " "    " "
## 4  ( 1 ) "*"  " "    " "     "*"       "*"     " "     " "    " "
## 5  ( 1 ) "*"  " "    " "     "*"       "*"     " "     " "    " "
## 6  ( 1 ) "*"  " "    " "     "*"       "*"     " "     " "    " "
## 7  ( 1 ) " "  " "    " "     "*"       "*"     " "     " "    " "
## 8  ( 1 ) " "  "*"    " "     "*"       "*"     " "     " "    " "
```

```
fit_all = regsubsets(Salary ~ ., data = Hitters, nvmax = 19)
fit_all_sum = summary(fit_all)
names(fit_all_sum)
```

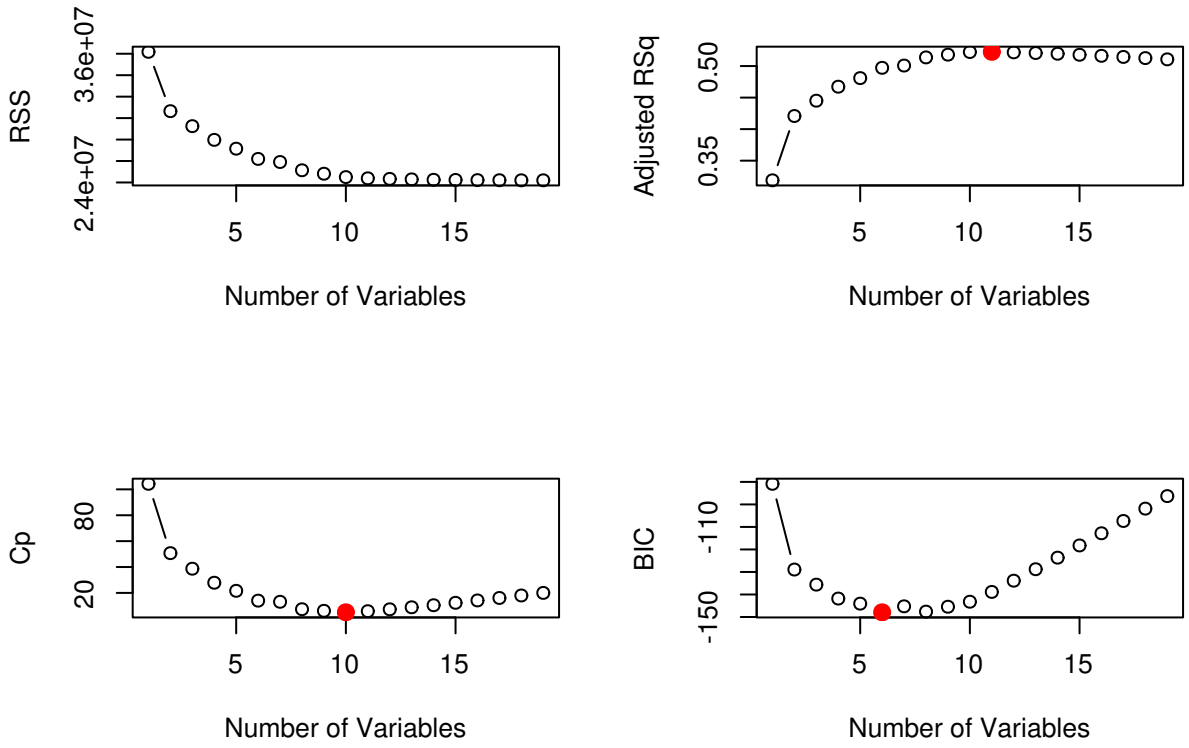
```
## [1] "which"  "rsq"    "rss"    "adjr2"  "cp"     "bic"    "outmat" "obj"
```

```
fit_all_sum$bic
```

```
##  [1]  -90.84637 -128.92622 -135.62693 -141.80892 -144.07143 -147.91690
```

```
##  [7] -145.25594 -147.61525 -145.44316 -143.21651 -138.86077 -133.87283
## [13] -128.77759 -123.64420 -118.21832 -112.81768 -107.35339 -101.86391
## [19]  -96.30412
```

```
par(mfrow = c(2, 2))
plot(fit_all_sum$rss, xlab = "Number of Variables", ylab = "RSS", type = "b")

plot(fit_all_sum$adjr2, xlab = "Number of Variables", ylab = "Adjusted RSq", type = "b")
best_adj_r2 = which.max(fit_all_sum$adjr2)
points(best_adj_r2, fit_all_sum$adjr2[best_adj_r2],
       col = "red",cex = 2, pch = 20)

plot(fit_all_sum$cp, xlab = "Number of Variables", ylab = "Cp", type = 'b')
best_cp = which.min(fit_all_sum$cp)
points(best_cp, fit_all_sum$cp[best_cp],
       col = "red", cex = 2, pch = 20)

plot(fit_all_sum$bic, xlab = "Number of Variables", ylab = "BIC", type = 'b')
best_bic = which.min(fit_all_sum$bic)
points(best_bic, fit_all_sum$bic[best_bic],
       col = "red", cex = 2, pch = 20)
```



22.1.3 Stepwise Methods

```
fit_fwd = regsubsets(Salary ~ ., data = Hitters, nvmax = 19, method = "forward")
fit_fwd_sum = summary(fit_fwd)
```

```
fit_bwd = regsubsets(Salary ~ ., data = Hitters, nvmax = 19, method = "backward")
fit_bwd_sum = summary(fit_bwd)
```

```
coef(fit_fwd, 7)
```

```
##  (Intercept)        AtBat         Hits        Walks         CRBI
##  109.7873062   -1.9588851    7.4498772    4.9131401    0.8537622
##       CWalks    DivisionW      PutOuts
##   -0.3053070 -127.1223928    0.2533404
```

```
coef(fit_bwd, 7)
```

```
##  (Intercept)        AtBat         Hits        Walks        CRuns
##  105.6487488   -1.9762838    6.7574914    6.0558691    1.1293095
##       CWalks    DivisionW      PutOuts
##   -0.7163346 -116.1692169    0.3028847
```

```
coef(fit_all, 7)
```

```
##  (Intercept)         Hits        Walks       CAtBat        CHits
##   79.4509472    1.2833513    3.2274264   -0.3752350    1.4957073
##       CHmRun    DivisionW      PutOuts
##    1.4420538 -129.9866432    0.2366813
```

```
fit_bwd_sum = summary(fit_bwd)
which.min(fit_bwd_sum$cp)
```

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

```
coef(fit_bwd, which.min(fit_bwd_sum$cp))
```

```
##  (Intercept)        AtBat         Hits        Walks       CAtBat
##  162.5354420   -2.1686501    6.9180175    5.7732246   -0.1300798
##        CRuns         CRBI       CWalks    DivisionW      PutOuts
##    1.4082490    0.7743122   -0.8308264 -112.3800575    0.2973726
##      Assists
##    0.2831680
```

```
fit = lm(Salary ~ ., data = Hitters)
fit_aic_back = step(fit, trace = FALSE)
coef(fit_aic_back)
```

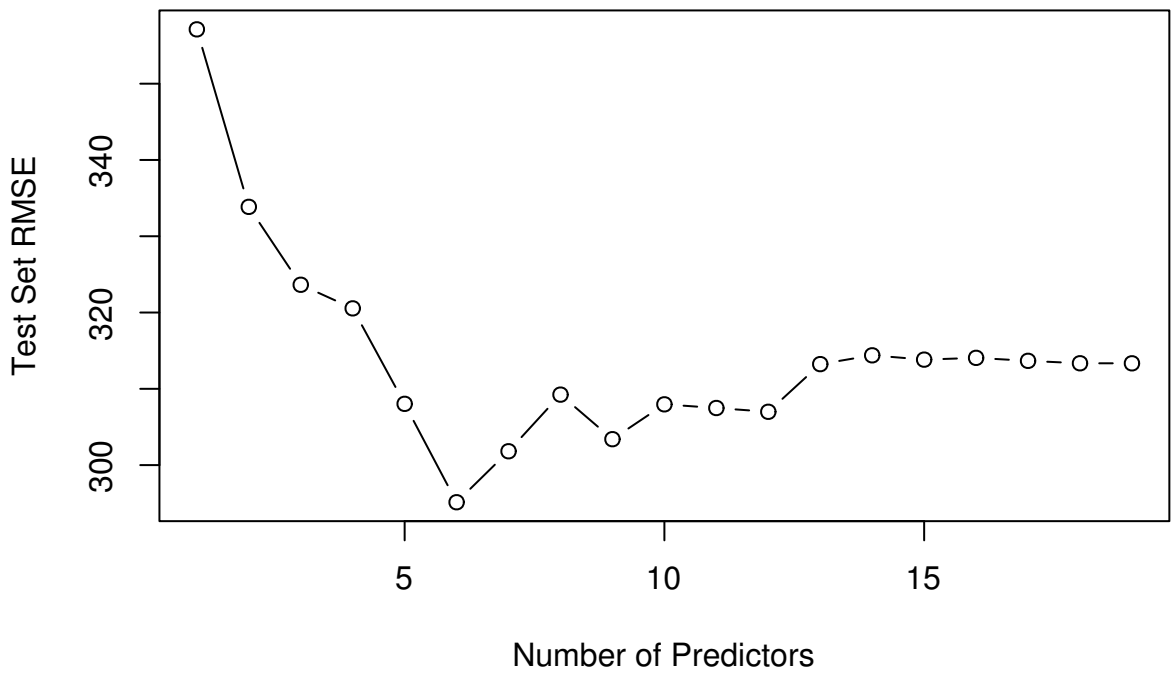
```
##  (Intercept)        AtBat         Hits        Walks       CAtBat
##  162.5354420   -2.1686501    6.9180175    5.7732246   -0.1300798
##        CRuns         CRBI       CWalks    DivisionW      PutOuts
##    1.4082490    0.7743122   -0.8308264 -112.3800575    0.2973726
##      Assists
##    0.2831680
```

22.2 Validated RMSE

```
set.seed(42)
num_vars = ncol(Hitters) - 1
trn_idx = sample(c(TRUE, FALSE), nrow(Hitters), rep = TRUE)
tst_idx = (!trn_idx)

fit_all = regsubsets(Salary ~ ., data = Hitters[trn_idx, ], nvmax = num_vars)
test_mat = model.matrix(Salary ~ ., data = Hitters[tst_idx, ])

test_err = rep(0, times = num_vars)
for (i in seq_along(test_err)) {
  coefs = coef(fit_all, id = i)
  pred = test_mat[, names(coefs)] %*% coefs
  test_err[i] <- sqrt(mean((Hitters$Salary[tst_idx] - pred) ^ 2))
}
test_err
```

```
##  [1] 357.1226 333.8531 323.6408 320.5458 308.0303 295.1308 301.8142
##  [8] 309.2389 303.3976 307.9660 307.4841 306.9883 313.2374 314.3905
## [15] 313.8258 314.0586 313.6674 313.3490 313.3424
```

```
plot(test_err, type='b', ylab = "Test Set RMSE", xlab = "Number of Predictors")
```



```
which.min(test_err)
```

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

```
coef(fit_all, which.min(test_err))
```

```
##  (Intercept)        Walks       CAtBat        CHits         CRBI
##  171.2082504    5.0067050   -0.4005457    1.2951923    0.7894534
##    DivisionW      PutOuts
## -131.1212694    0.2682166
```

```
class(fit_all)
```

```
## [1] "regsubsets"
```

```
predict.regsubsets = function(object, newdata, id, ...) {

  form  = as.formula(object$call[[2]])
  mat   = model.matrix(form, newdata)
  coefs = coef(object, id = id)
  xvars = names(coefs)

  mat[, xvars] %*% coefs
}
```

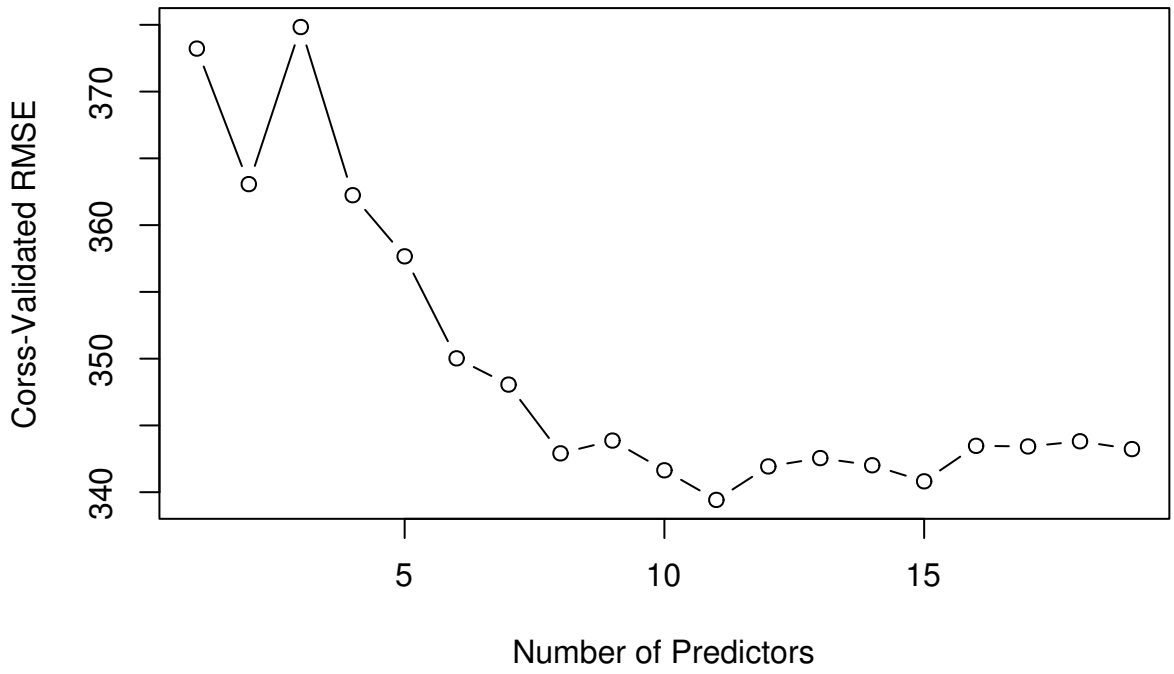
```
rmse = function(actual, predicted) {
  sqrt(mean((actual - predicted) ^ 2))
}
```

```
num_folds = 5
num_vars  = 19
set.seed(1)
folds = caret::createFolds(Hitters$Salary, k = num_folds)
fold_error = matrix(0, nrow = num_folds, ncol = num_vars,
                    dimnames = list(paste(1:5), paste(1:19)))

for(j in 1:num_folds) {

  train_fold    = Hitters[-folds[[j]], ]
  validate_fold = Hitters[ folds[[j]], ]


  best_fit = regsubsets(Salary ~ ., data = train_fold, nvmax = 19)

  for (i in 1:num_vars) {

    pred = predict(best_fit, validate_fold, id = i)

    fold_error[j, i] = rmse(actual = validate_fold$Salary,
                            predicted = pred)
  }
```

```
}

cv_error = apply(fold_error, 2, mean)
cv_error
```

```
##        1        2        3        4        5        6        7        8
## 373.2202 363.0715 374.8356 362.2405 357.6623 350.0238 348.0589 342.9089
##        9       10       11       12       13       14       15       16
## 343.8661 341.6405 339.4228 341.9303 342.5545 342.0155 340.8147 343.4722
##       17       18       19
## 343.4259 343.8129 343.2279
```

```
plot(cv_error, type='b', ylab = "Corss-Validated RMSE", xlab = "Number of Predictors")
```




```
fit_all = regsubsets(Salary ~ ., data = Hitters, nvmax = num_vars)
coef(fit_all, which.min(cv_error))
```

```
##  (Intercept)        AtBat         Hits        Walks       CAtBat
##  135.7512195   -2.1277482    6.9236994    5.6202755   -0.1389914
##        CRuns         CRBI       CWalks      LeagueN    DivisionW
##    1.4553310    0.7852528   -0.8228559   43.1116152 -111.1460252
##      PutOuts      Assists
##    0.2894087    0.2688277
```

22.3 External Links

- -

22.4 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
## [1] "leaps"
```

- Additional Packages, Not Attached

```
##  [1] "reshape2"     "kernlab"      "purrr"        "splines"
##  [5] "lattice"      "colorspace"   "stats4"       "htmltools"
##  [9] "yaml"         "survival"     "prodlim"      "rlang"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "foreach"      "plyr"         "bindr"        "dimRed"
## [21] "lava"         "robustbase"   "stringr"      "timeDate"
## [25] "munsell"      "gtable"       "recipes"      "codetools"
## [29] "evaluate"     "knitr"        "caret"        "class"
## [33] "DEoptimR"     "methods"      "Rcpp"         "scales"
## [37] "backports"    "ipred"        "CVST"         "ggplot2"
## [41] "digest"       "stringi"      "bookdown"     "dplyr"
## [45] "RcppRoll"     "ddalpha"      "grid"         "rprojroot"
## [49] "tools"        "magrittr"     "lazyeval"     "tibble"
## [53] "DRR"          "pkgconfig"    "MASS"         "Matrix"
## [57] "lubridate"    "gower"        "assertthat"   "rmarkdown"
## [61] "iterators"    "R6"           "rpart"        "nnet"
## [65] "nlme"         "compiler"
```

Part VI

The Modern Era

Chapter 23

Overview

Chapter 24

Regularization

TODO: Introduce regularization as a concept.

We will use the `Hitters` dataset from the `ISLR` package to explore two shrinkage methods: **ridge** and **lasso**. These are otherwise known as **penalized regression** methods.

```
data(Hitters, package = "ISLR")
```

This dataset has some missing data in the response `Salaray`. We use the `na.omit()` function the clean the dataset.

```
sum(is.na(Hitters))
```

```
## [1] 59
```

```
sum(is.na(Hitters$Salary))
```

```
## [1] 59
```

```
Hitters = na.omit(Hitters)
sum(is.na(Hitters))
```

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

The predictors variables are offensive and defensive statistics for a number of baseball players.

```
names(Hitters)
```

```
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"
##  [6] "Walks"     "Years"     "CAtBat"    "CHits"     "CHmRun"
## [11] "CRuns"     "CRBI"      "CWalks"    "League"    "Division"
## [16] "PutOuts"   "Assists"   "Errors"    "Salary"    "NewLeague"
```

We use the `glmnet()` and `cv.glmnet()` functions in the `glmnet` package to fit penalized regressions.

```
# this is a temporary workaround for an issue with glmnet, Matrix, and R version 3.3.3
# see here: http://stackoverflow.com/questions/43282720/r-error-in-validobject-object-when-running-as-s
library(methods)
```

```
library(glmnet)
```

The `glmnet` function does not allow the use of model formulas, so we setup the data for ease of use with `glmnet`.

```
X = model.matrix(Salary ~ ., Hitters)[, -1]
y = Hitters$Salary
```

First, we fit a regular linear regression, and note the size of the predictors' coefficients, and predictors' coefficients squared. (The two penalties we will use.)

```
fit = lm(Salary ~ ., Hitters)
coef(fit)
```

```
##  (Intercept)        AtBat         Hits        HmRun         Runs
##  163.1035878   -1.9798729    7.5007675    4.3308829   -2.3762100
##          RBI        Walks        Years       CAtBat        CHits
##   -1.0449620    6.2312863   -3.4890543   -0.1713405    0.1339910
##       CHmRun        CRuns         CRBI       CWalks      LeagueN
##   -0.1728611    1.4543049    0.8077088   -0.8115709   62.5994230
##    DivisionW      PutOuts      Assists       Errors   NewLeagueN
## -116.8492456    0.2818925    0.3710692   -3.3607605  -24.7623251
```

```
sum(abs(coef(fit)[-1]))
```

```
## [1] 238.7295
```

```
sum(coef(fit)[-1] ^ 2)
```

```
## [1] 18337.3
```

24.1 Ridge Regression

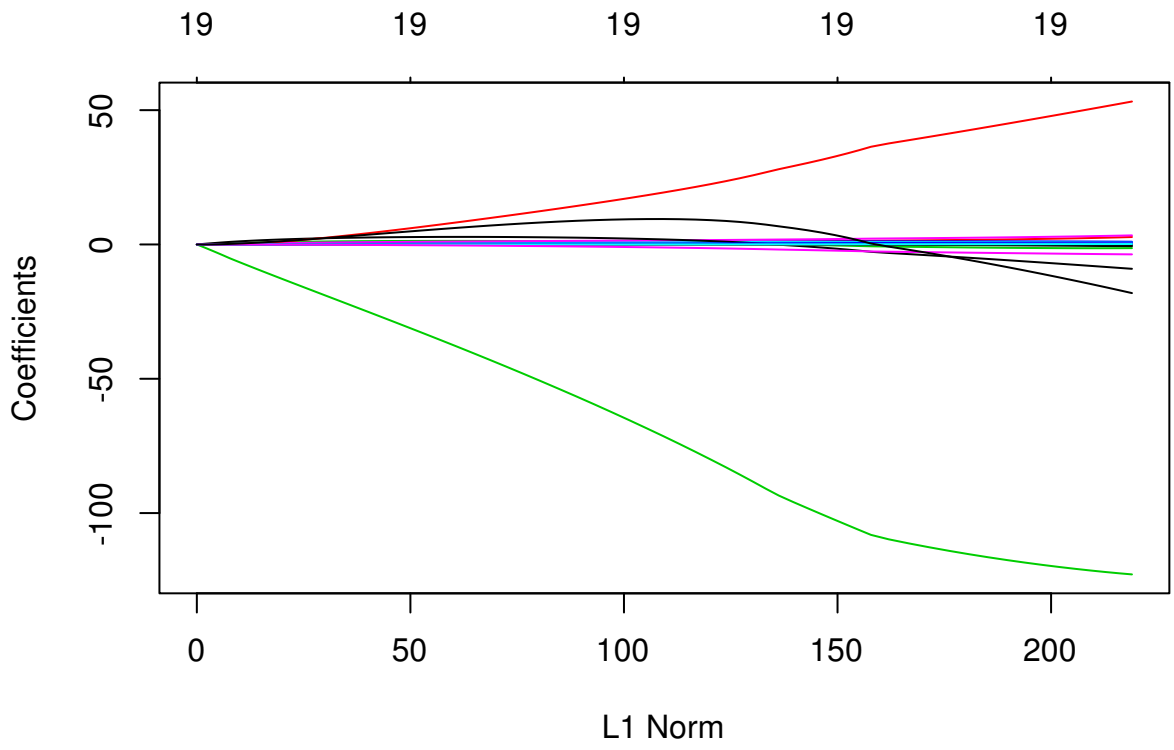
We first illustrate **ridge regression**, which can be fit using `glmnet()` with `alpha = 0` and seeks to minimize

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2.$$

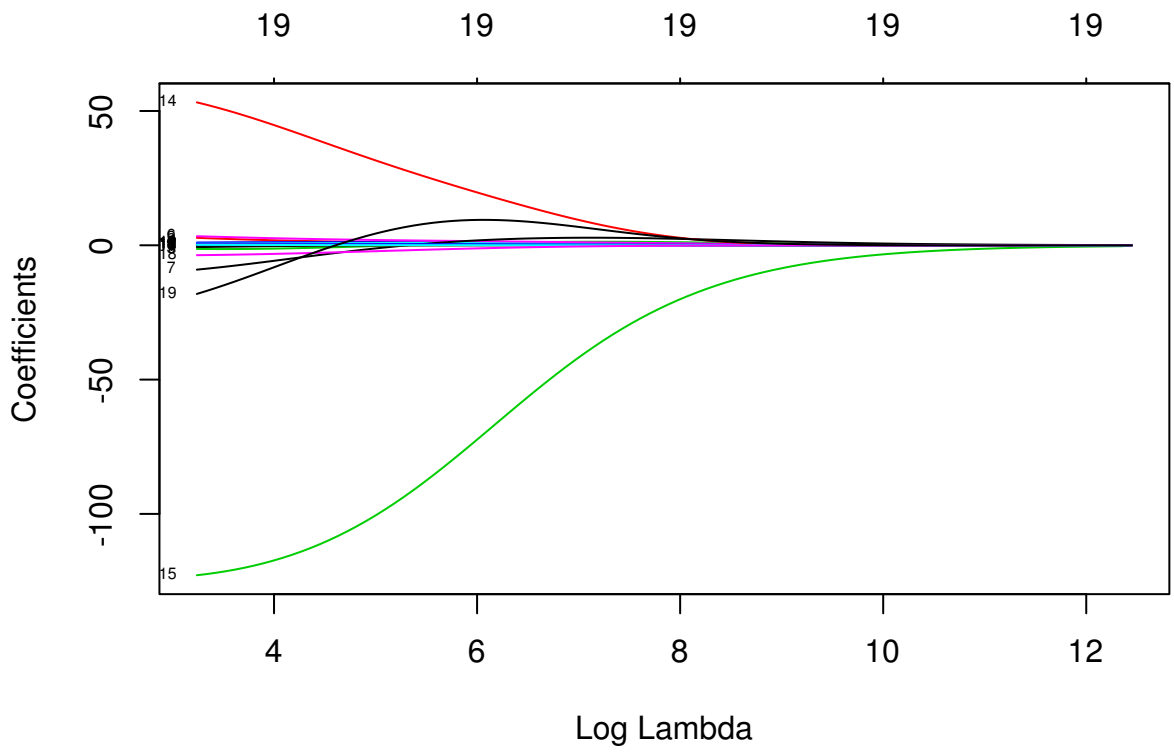
Notice that the intercept is **not** penalized. Also, note that that ridge regression is **not** scale invariant like the usual unpenalized regression. Thankfully, `glmnet()` takes care of this internally. It automatically standardizes input for fitting, then reports fitted coefficient using the original scale.

The two plots illustrate how much the coefficients are penalized for different values of λ. Notice none of the coefficients are forced to be zero.

```
fit_ridge = glmnet(X, y, alpha = 0)
plot(fit_ridge)
```




```
plot(fit_ridge, xvar = "lambda", label = TRUE)
```

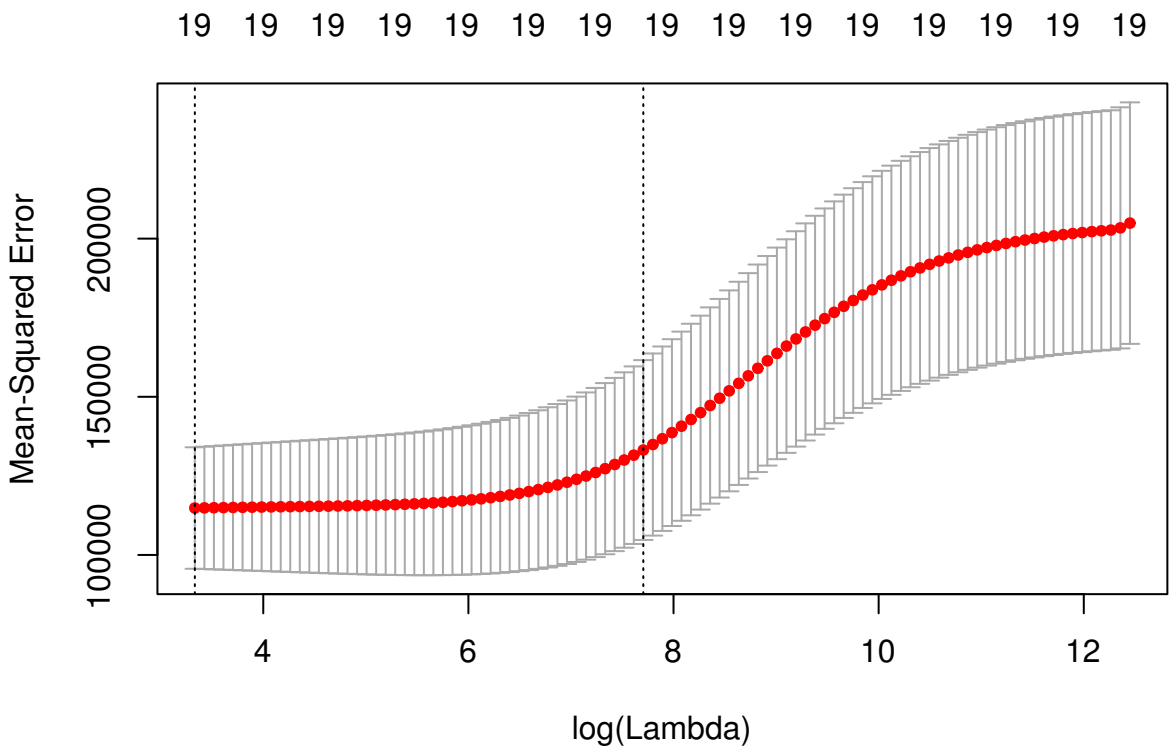



```
dim(coef(fit_ridge))
```

```
## [1]  20 100
```

We use cross-validation to select a good λ value. The `cv.glmnet()`function uses 10 folds by default. The plot illustrates the MSE for the λs considered. Two lines are drawn. The first is the λ that gives the smallest MSE. The second is the λ that gives an MSE within one standard error of the smallest.

```
fit_ridge_cv = cv.glmnet(X, y, alpha = 0)
plot(fit_ridge_cv)
```



The `cv.glmnet()` function returns several details of the fit for both λ values in the plot. Notice the penalty terms are smaller than the full linear regression. (As we would expect.)

```
coef(fit_ridge_cv)
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept) 185.946731847
## AtBat         0.096634022
## Hits          0.408580478
## HmRun         1.242303539
## Runs          0.650047295
## RBI           0.642033635
## Walks         0.848737422
## Years         2.608433226
## CAtBat        0.008188531
## CHits         0.031829975
## CHmRun        0.235663247
## CRuns         0.063816873
## CRBI          0.066045116
## CWalks        0.062642350
## LeagueN       4.252099497
## DivisionW   -25.296959330
## PutOuts       0.059902888
## Assists       0.008305300
## Errors       -0.185603402
```

```
## NewLeagueN    3.676189338
```

```
coef(fit_ridge_cv, s = "lambda.min")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                           1
## (Intercept)  7.645824e+01
## AtBat       -6.315180e-01
## Hits         2.642160e+00
## HmRun       -1.388233e+00
## Runs         1.045729e+00
## RBI          7.315713e-01
## Walks        3.278001e+00
## Years       -8.723734e+00
## CAtBat       1.256354e-04
## CHits        1.318975e-01
## CHmRun       6.895578e-01
## CRuns        2.830055e-01
## CRBI         2.514905e-01
## CWalks      -2.599851e-01
## LeagueN      5.233720e+01
## DivisionW   -1.224170e+02
## PutOuts      2.623667e-01
## Assists      1.629044e-01
## Errors      -3.644002e+00
## NewLeagueN  -1.702598e+01
```

```
sum(coef(fit_ridge_cv, s = "lambda.min")[-1] ^ 2) # penalty term for lambda minimum
```

```
## [1] 18126.85
```

```
coef(fit_ridge_cv, s = "lambda.1se")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                           1
## (Intercept) 185.946731847
## AtBat         0.096634022
## Hits          0.408580478
## HmRun         1.242303539
## Runs          0.650047295
## RBI           0.642033635
## Walks         0.848737422
## Years         2.608433226
## CAtBat        0.008188531
## CHits         0.031829975
## CHmRun        0.235663247
## CRuns         0.063816873
## CRBI          0.066045116
## CWalks        0.062642350
## LeagueN       4.252099497
## DivisionW   -25.296959330
## PutOuts       0.059902888
```

```
## Assists       0.008305300
## Errors       -0.185603402
## NewLeagueN    3.676189338
```

```
sum(coef(fit_ridge_cv, s = "lambda.1se")[-1] ^ 2) # penalty term for lambda one SE
```

```
## [1] 681.7166
```

```
#predict(fit_ridge_cv, X, s = "lambda.min")
#predict(fit_ridge_cv, X)
mean((y - predict(fit_ridge_cv, X)) ^ 2) # "train error"
```

```
## [1] 128551
```

```
sqrt(fit_ridge_cv$cvm) # CV-RMSEs
```

```
##  [1] 452.6564 450.9968 450.2320 449.9641 449.6713 449.3513 449.0018
##  [8] 448.6202 448.2037 447.7496 447.2545 446.7153 446.1283 445.4899
## [15] 444.7961 444.0426 443.2254 442.3401 441.3820 440.3464 439.2287
## [22] 438.0243 436.7285 435.3368 433.8451 432.2496 430.5467 428.7338
## [29] 426.8087 424.7701 422.6178 420.3528 417.9772 415.4947 412.9103
## [36] 410.2310 407.4651 404.6225 401.7148 398.7552 395.7582 392.7393
## [43] 389.7150 386.7022 383.7180 380.7792 377.9022 375.1022 372.3930
## [50] 369.7869 367.2940 364.9230 362.6803 360.5702 358.5947 356.7542
## [57] 355.0472 353.4707 352.0206 350.6915 349.4776 348.3723 347.3689
## [64] 346.4600 345.6406 344.9057 344.2457 343.6540 343.1252 342.6566
## [71] 342.2410 341.8742 341.5511 341.2692 341.0185 340.8016 340.6144
## [78] 340.4490 340.3070 340.1817 340.0738 339.9765 339.8922 339.8148
## [85] 339.7439 339.6787 339.6169 339.5567 339.4980 339.4391 339.3799
## [92] 339.3197 339.2586 339.1954 339.1325 339.0672 339.0026 338.9386
## [99] 338.8744
```

```
sqrt(fit_ridge_cv$cvm[fit_ridge_cv$lambda == fit_ridge_cv$lambda.min]) # CV-RMSE minimum
```

```
## [1] 338.8744
```

```
sqrt(fit_ridge_cv$cvm[fit_ridge_cv$lambda == fit_ridge_cv$lambda.1se]) # CV-RMSE one SE
```

```
## [1] 364.923
```

24.2 Lasso

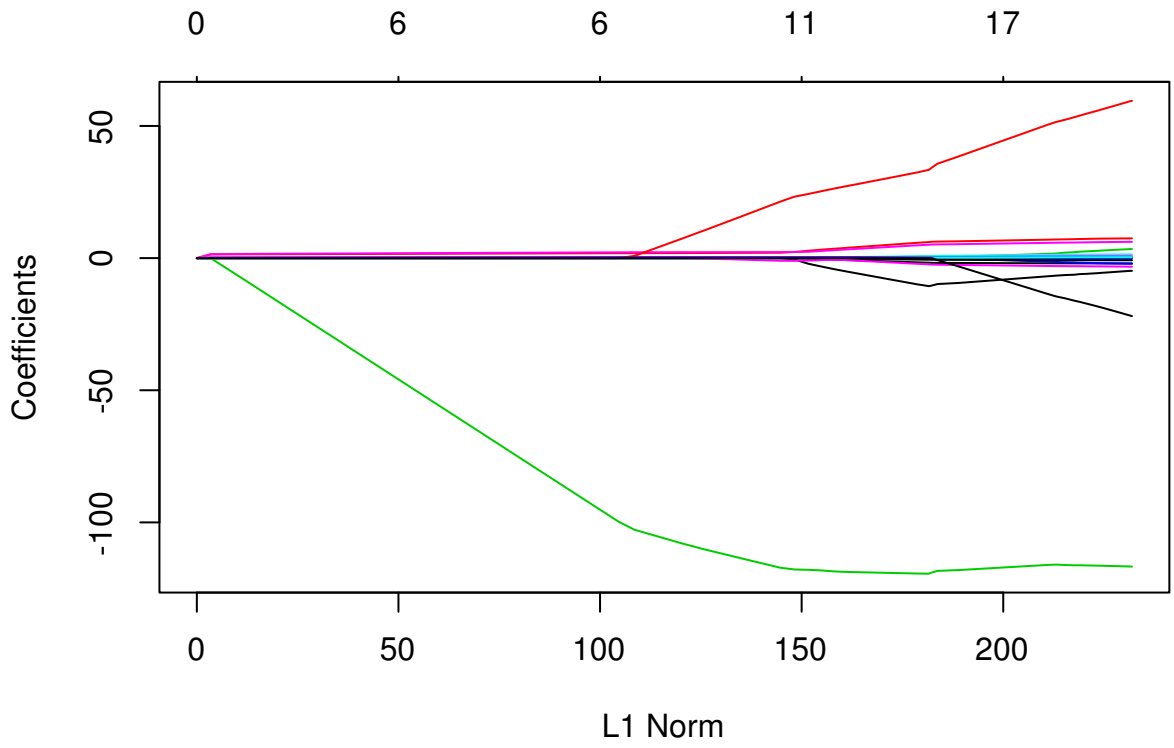
We now illustrate **lasso**, which can be fit using `glmnet()` with `alpha = 1` and seeks to minimize

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|.$$

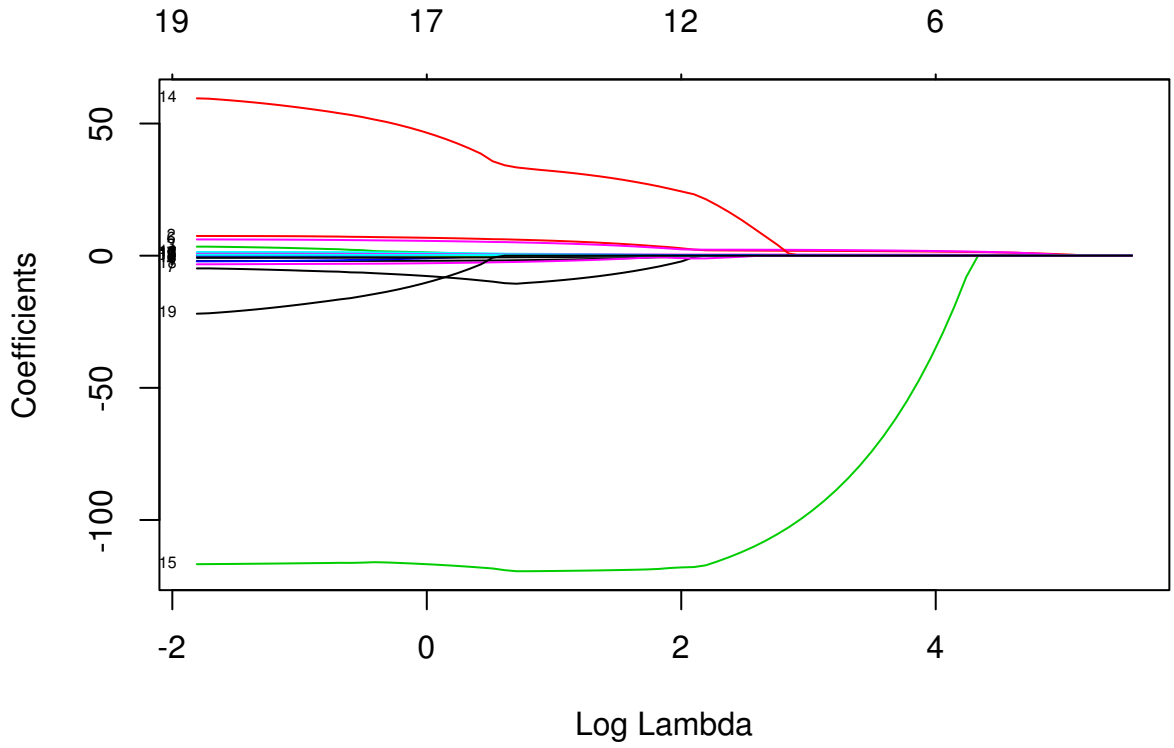
Like ridge, lasso is not scale invariant.

The two plots illustrate how much the coefficients are penalized for different values of λ. Notice some of the coefficients are forced to be zero.

```
fit_lasso = glmnet(X, y, alpha = 1)
plot(fit_lasso)
```




```
plot(fit_lasso, xvar = "lambda", label = TRUE)
```

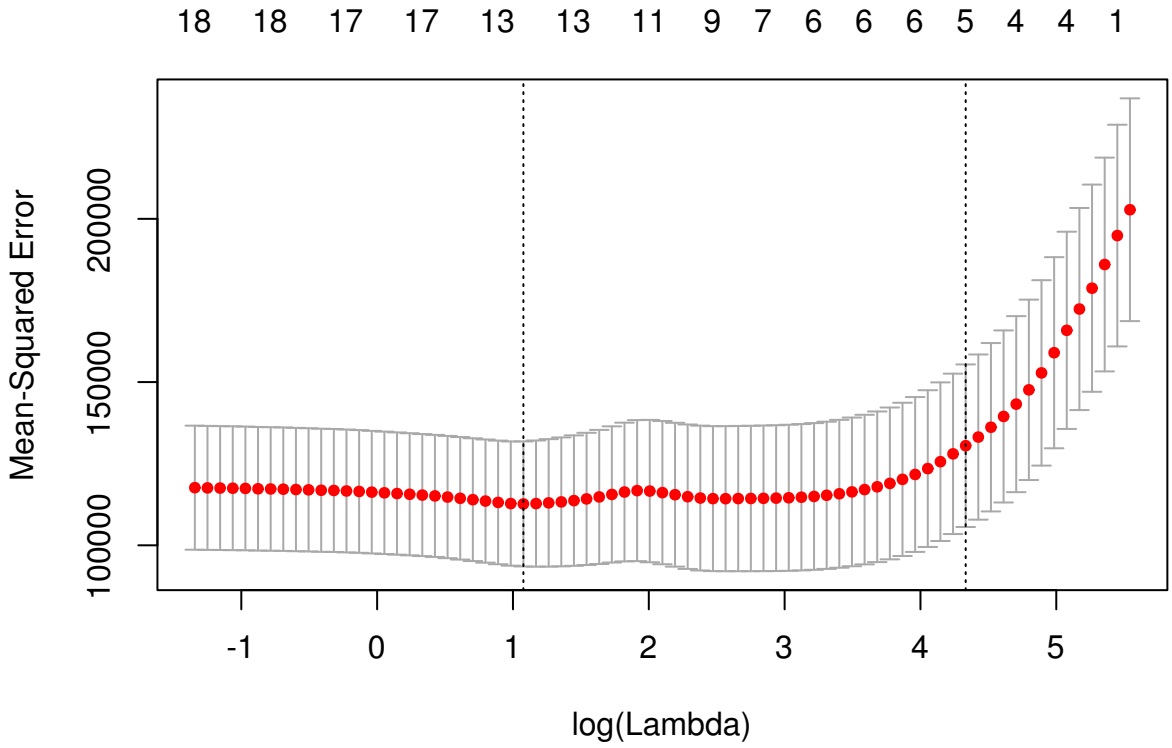



```
dim(coef(fit_lasso))
```

```
## [1] 20 80
```

Again, to actually pick a λ, we will use cross-validation. The plot is similar to the ridge plot. Notice along the top is the number of features in the model. (Which changed in this plot.)

```
fit_lasso_cv = cv.glmnet(X, y, alpha = 1)
plot(fit_lasso_cv)
```



`cv.glmnet()` returns several details of the fit for both λ values in the plot. Notice the penalty terms are again smaller than the full linear regression. (As we would expect.) Some coefficients are 0.

```
coef(fit_lasso_cv)
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept) 144.37970458
## AtBat         .
## Hits          1.36380384
## HmRun         .
## Runs          .
## RBI           .
## Walks         1.49731098
## Years         .
## CAtBat        .
## CHits         .
## CHmRun        .
## CRuns         0.15275165
## CRBI          0.32833941
## CWalks        .
## LeagueN       .
## DivisionW     .
## PutOuts       0.06625755
## Assists       .
## Errors        .
```

```
## NewLeagueN    .
```

```
coef(fit_lasso_cv, s = "lambda.min")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept)  117.5258439
## AtBat         -1.4742901
## Hits           5.4994256
## HmRun          .
## Runs           .
## RBI            .
## Walks          4.5991651
## Years         -9.1918308
## CAtBat         .
## CHits          .
## CHmRun         0.4806743
## CRuns          0.6354799
## CRBI           0.3956153
## CWalks        -0.4993240
## LeagueN       31.6238174
## DivisionW   -119.2516409
## PutOuts        0.2704287
## Assists        0.1594997
## Errors        -1.9426357
## NewLeagueN     .
```

```
sum(abs(coef(fit_lasso_cv, s = "lambda.min")[-1])) # penalty term for lambda minimum
```

```
## [1] 176.0238
```

```
coef(fit_lasso_cv, s = "lambda.1se")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                         1
## (Intercept) 144.37970458
## AtBat          .
## Hits           1.36380384
## HmRun          .
## Runs           .
## RBI            .
## Walks          1.49731098
## Years          .
## CAtBat         .
## CHits          .
## CHmRun         .
## CRuns          0.15275165
## CRBI           0.32833941
## CWalks         .
## LeagueN        .
## DivisionW      .
## PutOuts        0.06625755
```

```
## Assists       .
## Errors        .
## NewLeagueN    .
```

```
sum(abs(coef(fit_lasso_cv, s = "lambda.1se")[-1])) # penalty term for lambda one SE
```

```
## [1] 3.408463
```

```
#predict(fit_lasso_cv, X, s = "lambda.min")
#predict(fit_lasso_cv, X)
mean((y - predict(fit_lasso_cv, X)) ^ 2) # "train error"
```

```
## [1] 121290.9
```

```
sqrt(fit_lasso_cv$cvm)
```

```
##  [1] 450.3228 441.4492 431.2887 422.7981 415.1785 407.2484 398.7369
##  [8] 390.9078 384.2152 378.4727 373.4599 369.0101 364.9040 361.2576
## [15] 357.7893 354.4198 351.4229 348.8458 346.6937 344.8989 343.4015
## [22] 342.1545 341.1174 340.2701 339.5857 339.0694 338.7137 338.4680
## [29] 338.3028 338.2077 338.1431 338.1058 338.0506 338.0746 338.3434
## [36] 338.9465 339.8521 340.7859 341.4840 341.6905 341.0559 339.9710
## [43] 338.8451 337.9531 337.1877 336.5962 336.1522 335.8261 335.6668
## [50] 335.8431 336.3625 336.9754 337.6084 338.2207 338.7925 339.3071
## [57] 339.6849 340.0470 340.3752 340.6976 341.0050 341.3162 341.5322
## [64] 341.7544 341.8459 342.0429 342.1688 342.3308 342.4303 342.5076
## [71] 342.6712 342.7563 342.8310 342.9097 343.0038
```

```
sqrt(fit_lasso_cv$cvm[fit_lasso_cv$lambda == fit_lasso_cv$lambda.min]) # CV-RMSE minimum
```

```
## [1] 335.6668
```

```
sqrt(fit_lasso_cv$cvm[fit_lasso_cv$lambda == fit_lasso_cv$lambda.1se]) # CV-RMSE one SE
```

```
## [1] 361.2576
```

24.3 broom

Sometimes, the output from `glmnet()` can be overwhelming. The `broom` package can help with that.

```
library(broom)
#fit_lasso_cv
tidy(fit_lasso_cv)
```

```
##         lambda estimate std.error conf.high  conf.low nzero
## 1  255.2820965 202790.6  34117.91  236908.6 168672.73     0
## 2  232.6035386 194877.4  33953.52  228830.9 160923.87     1
## 3  211.9396813 186009.9  32741.96  218751.9 153267.95     2
```

```
## 4  193.1115442 178758.2  31732.58  210490.8 147025.62     2
## 5  175.9560468 172373.2  30954.40  203327.6 141418.82     3
## 6  160.3245966 165851.2  30205.99  196057.2 135645.25     4
## 7  146.0818013 158991.1  29270.89  188262.0 129720.23     4
## 8  133.1042967 152808.9  28394.34  181203.2 124414.55     4
## 9  121.2796778 147621.3  27629.80  175251.1 119991.52     4
## 10 110.5055255 143241.6  26961.62  170203.2 116279.98     4
## 11 100.6885192 139472.3  26351.58  165823.9 113120.69     5
## 12  91.7436287 136168.4  25784.83  161953.3 110383.59     5
## 13  83.5933775 133155.0  25297.80  158452.8 107857.17     5
## 14  76.1671723 130507.1  24896.21  155403.3 105610.85     5
## 15  69.4006906 128013.2  24563.18  152576.3 103449.98     6
## 16  63.2353245 125613.4  24292.86  149906.3 101320.57     6
## 17  57.6176726 123498.0  24005.59  147503.6  99492.45     6
## 18  52.4990774 121693.4  23717.71  145411.1  97975.67     6
## 19  47.8352040 120196.5  23469.26  143665.8  96727.24     6
## 20  43.5856563 118955.3  23255.06  142210.3  95700.20     6
## 21  39.7136268 117924.6  23069.87  140994.4  94854.70     6
## 22  36.1855776 117069.7  22909.91  139979.6  94159.80     6
## 23  32.9709506 116361.1  22771.50  139132.6  93589.56     6
## 24  30.0419022 115783.7  22649.21  138432.9  93134.52     6
## 25  27.3730624 115318.4  22546.47  137864.9  92771.97     6
## 26  24.9413150 114968.0  22466.67  137434.7  92501.36     6
## 27  22.7255973 114727.0  22413.73  137140.7  92313.24     6
## 28  20.7067179 114560.6  22370.76  136931.3  92189.81     6
## 29  18.8671902 114448.8  22333.43  136782.2  92115.36     6
## 30  17.1910810 114384.5  22301.84  136686.3  92082.64     7
## 31  15.6638727 114340.8  22273.29  136614.1  92067.48     7
## 32  14.2723374 114315.5  22248.00  136563.5  92067.54     7
## 33  13.0044223 114278.2  22222.18  136500.4  92056.01     9
## 34  11.8491453 114294.4  22199.26  136493.7  92095.16     9
## 35  10.7964999 114476.3  22177.09  136653.4  92299.19     9
## 36   9.8373686 114884.7  22119.98  137004.7  92764.73     9
## 37   8.9634439 115499.4  22027.91  137527.3  93471.51     9
## 38   8.1671562 116135.0  21937.82  138072.9  94197.22    11
## 39   7.4416086 116611.3  21810.50  138421.8  94800.83    11
## 40   6.7805166 116752.4  21595.66  138348.0  95156.71    12
## 41   6.1781542 116319.1  21244.35  137563.5  95074.79    12
## 42   5.6293040 115580.3  20859.66  136439.9  94720.61    13
## 43   5.1292121 114816.0  20538.64  135354.7  94277.39    13
## 44   4.6735471 114212.3  20247.76  134460.0  93964.54    13
## 45   4.2583620 113695.5  19980.52  133676.1  93715.00    13
## 46   3.8800609 113297.0  19739.66  133036.6  93557.31    13
## 47   3.5353670 112998.3  19523.78  132522.1  93474.52    13
## 48   3.2212947 112779.2  19322.80  132102.0  93456.39    13
## 49   2.9351238 112672.2  19146.40  131818.6  93525.79    13
## 50   2.6743755 112790.6  19021.33  131811.9  93769.28    13
## 51   2.4367913 113139.8  18934.04  132073.8  94205.73    13
## 52   2.2203135 113552.4  18862.02  132414.4  94690.40    14
## 53   2.0230670 113979.5  18805.62  132785.1  95173.83    15
## 54   1.8433433 114393.2  18760.90  133154.1  95632.32    15
## 55   1.6795857 114780.3  18724.07  133504.4  96056.27    17
## 56   1.5303760 115129.3  18690.37  133819.7  96438.97    17
## 57   1.3944216 115385.9  18688.56  134074.4  96697.30    17
```

```
## 58   1.2705450 115632.0  18701.15  134333.1  96930.84    17
## 59   1.1576733 115855.2  18722.38  134577.6  97132.87    17
## 60   1.0548288 116074.9  18746.75  134821.6  97328.11    17
## 61   0.9611207 116284.4  18769.50  135053.9  97514.92    17
## 62   0.8757374 116496.8  18792.14  135288.9  97704.64    17
## 63   0.7979393 116644.3  18814.21  135458.5  97830.05    17
## 64   0.7270526 116796.1  18831.21  135627.3  97964.88    17
## 65   0.6624632 116858.6  18850.01  135708.6  98008.58    18
## 66   0.6036118 116993.3  18868.66  135862.0  98124.67    18
## 67   0.5499886 117079.5  18892.24  135971.7  98187.26    18
## 68   0.5011291 117190.4  18908.15  136098.5  98282.22    17
## 69   0.4566102 117258.5  18919.10  136177.6  98339.40    18
## 70   0.4160462 117311.5  18934.04  136245.5  98377.44    18
## 71   0.3790858 117423.5  18946.78  136370.3  98476.76    18
## 72   0.3454089 117481.8  18965.71  136447.6  98516.14    18
## 73   0.3147237 117533.1  18975.50  136508.6  98557.60    18
## 74   0.2867645 117587.1  18990.30  136577.4  98596.80    18
## 75   0.2612891 117651.6  18998.45  136650.1  98653.16    18
```

```
glance(fit_lasso_cv) # the two lambda values of interest
```

```
##   lambda.min lambda.1se
## 1   2.935124   76.16717
```

24.4 Simulation Study, p > n

Aside from simply shrinking coefficients (ridge) and setting some coefficients to 0 (lasso), penalized regression also has the advantage of being able to handle the $p > n$ case.

```
set.seed(1234)
n = 1000
p = 5500
X = replicate(p, rnorm(n = n))
beta = c(1, 1, 1, rep(0, 5497))
z = X %*% beta
prob = exp(z) / (1 + exp(z))
y = as.factor(rbinom(length(z), size = 1, prob = prob))
```

We first simulate a classification example where $p > n$.

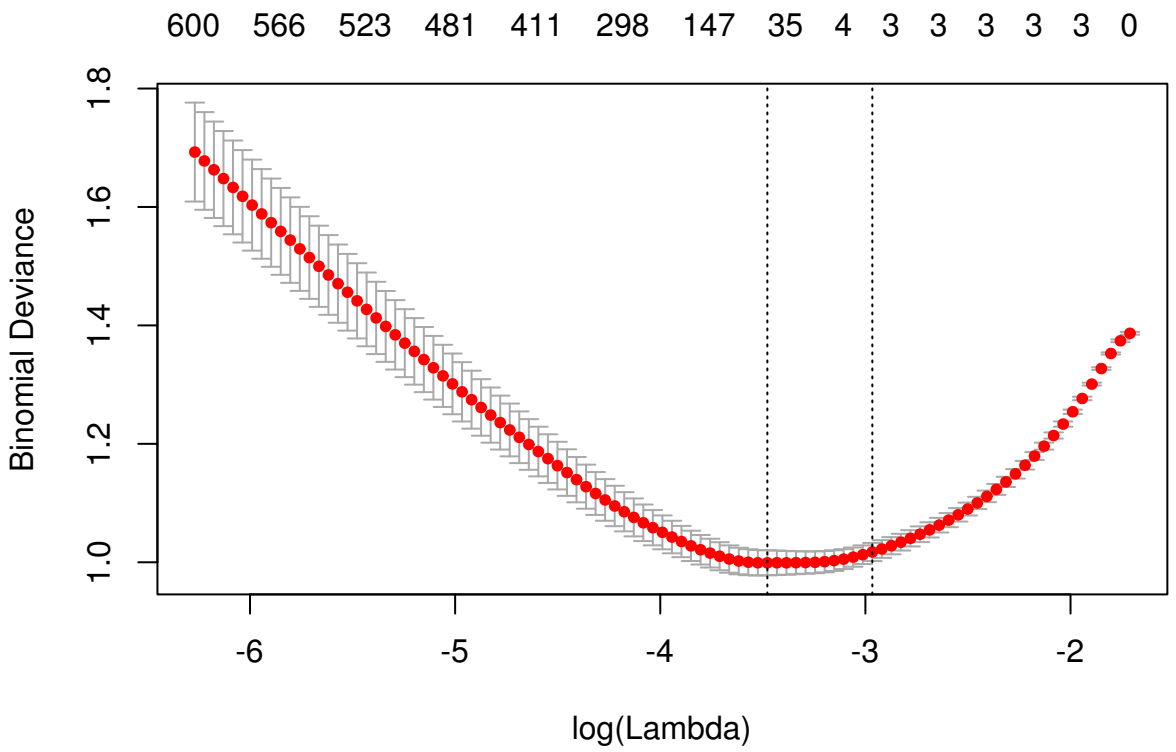
```
# glm(y ~ X, family = "binomial")
# will not converge
```

We then use a lasso penalty to fit penalized logistic regression. This minimizes

$$\sum_{i=1}^{n} L\left(y_i, \beta_0 + \sum_{j=1}^{p} \beta_j x_{ij}\right) + \lambda \sum_{j=1}^{p} |\beta_j|$$

where L is the appropriate *negative* **log**-likelihood.

```
library(glmnet)
fit_cv = cv.glmnet(X, y, family = "binomial", alpha = 1)
plot(fit_cv)
```




```
head(coef(fit_cv), n = 10)
```

```
## 10 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept) 0.02397452
## V1          0.59674958
## V2          0.56251761
## V3          0.60065105
## V4          .
## V5          .
## V6          .
## V7          .
## V8          .
## V9          .
```

```
fit_cv$nzero
```

```
##  s0  s1  s2  s3  s4  s5  s6  s7  s8  s9 s10 s11 s12 s13 s14 s15 s16 s17
##   0   2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3
## s18 s19 s20 s21 s22 s23 s24 s25 s26 s27 s28 s29 s30 s31 s32 s33 s34 s35
```

```
##   3   3   3   3   3   3   3   3   3   3   3   3   4   6   7  10  18  24
## s36 s37 s38 s39 s40 s41 s42 s43 s44 s45 s46 s47 s48 s49 s50 s51 s52 s53
##  35  54  65  75  86 100 110 129 147 168 187 202 221 241 254 269 283 298
## s54 s55 s56 s57 s58 s59 s60 s61 s62 s63 s64 s65 s66 s67 s68 s69 s70 s71
## 310 324 333 350 364 375 387 400 411 429 435 445 453 455 462 466 475 481
## s72 s73 s74 s75 s76 s77 s78 s79 s80 s81 s82 s83 s84 s85 s86 s87 s88 s89
## 487 491 496 498 502 504 512 518 523 526 528 536 543 550 559 561 563 566
## s90 s91 s92 s93 s94 s95 s96 s97 s98
## 570 571 576 582 586 590 596 596 600
```

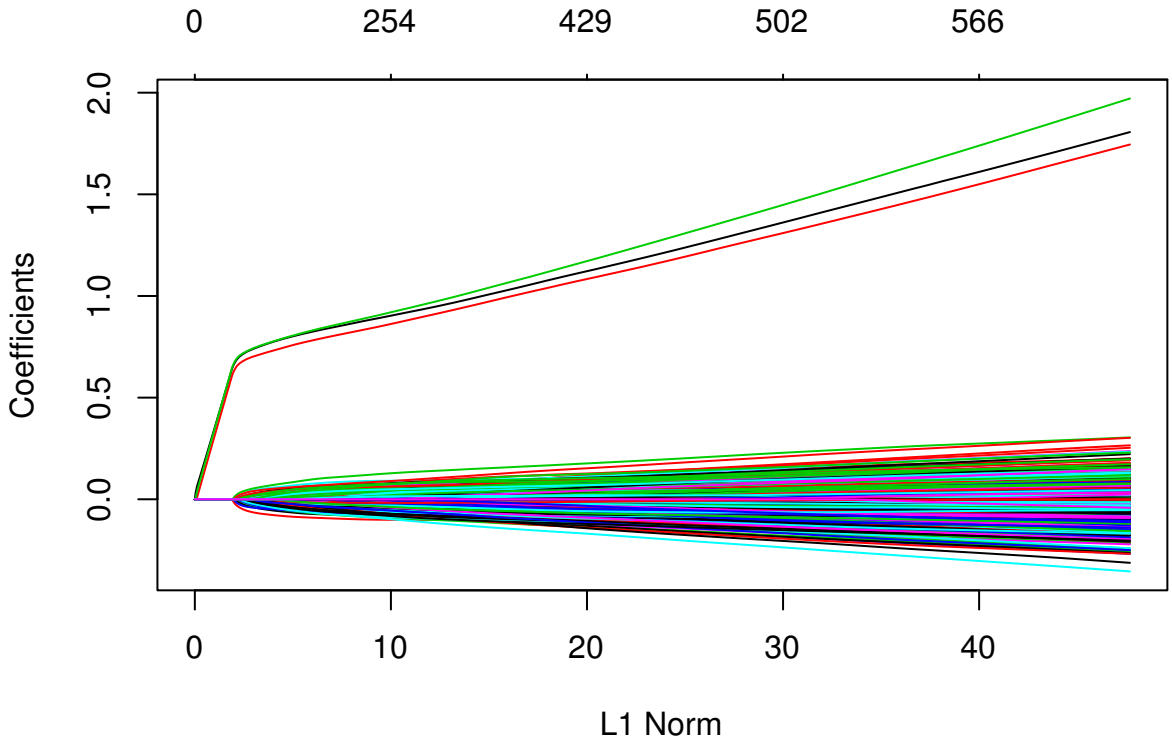
Notice, only the first three predictors generated are truly significant, and that is exactly what the suggested model finds.

```
fit_1se = glmnet(X, y, family = "binomial", lambda = fit_cv$lambda.1se)
which(as.vector(as.matrix(fit_1se$beta)) != 0)
```

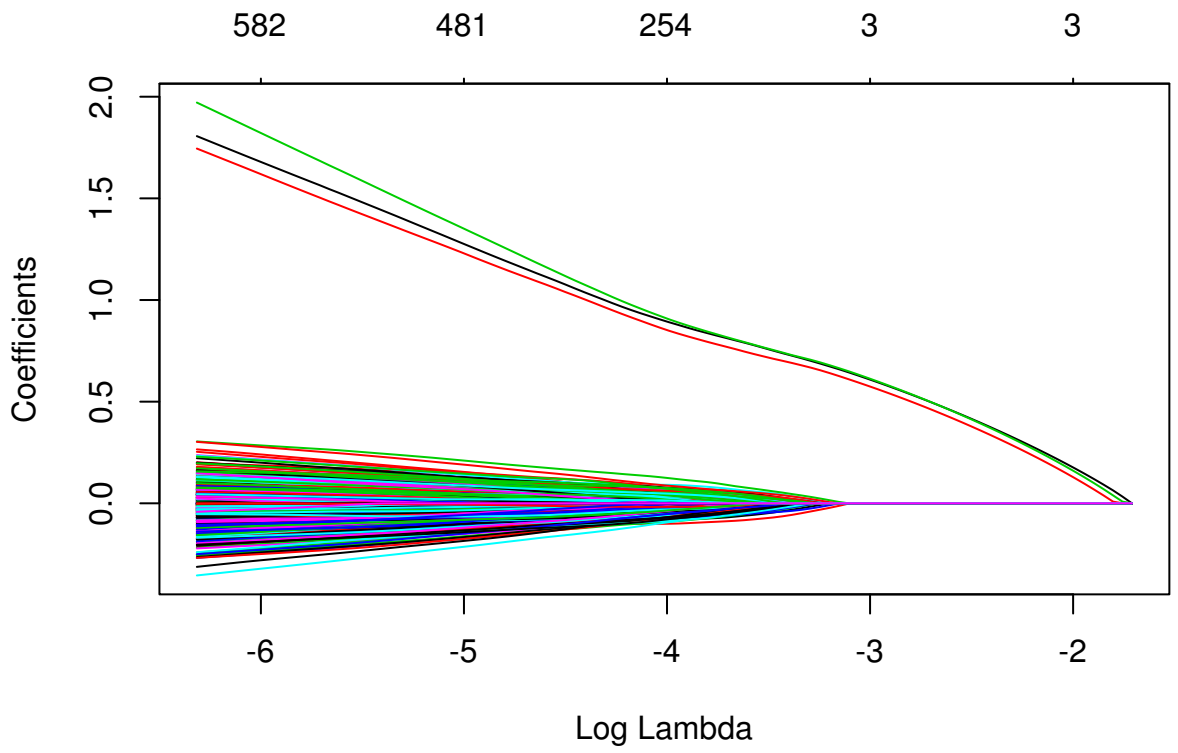
```
## [1] 1 2 3
```

We can also see in the following plots, the three features entering the model well ahead of the irrelevant features.

```
plot(glmnet(X, y, family = "binomial"))
```



```
plot(glmnet(X, y, family = "binomial"), xvar = "lambda")
```



We can extract the two relevant λ values.

```
fit_cv$lambda.min
```

```
## [1] 0.03087158
```

```
fit_cv$lambda.1se
```

```
## [1] 0.0514969
```

Since `cv.glmnet()` does not calculate prediction accuracy for classification, we take the λ values and create a grid for `caret` to search in order to obtain prediction accuracy with `train()`. We set $\alpha = 1$ in this grid, as `glmnet` can actually tune over the $\alpha = 1$ parameter. (More on that later.)

Note that we have to force `y` to be a factor, so that `train()` recognizes we want to have a binomial response. The `train()` function in `caret` use the type of variable in `y` to determine if you want to use `family = "binomial"` or `family = "gaussian"`.

```
library(caret)
cv_5 = trainControl(method = "cv", number = 5)
lasso_grid = expand.grid(alpha = 1,
                         lambda = c(fit_cv$lambda.min, fit_cv$lambda.1se))
lasso_grid
```

```
##   alpha     lambda
## 1     1 0.03087158
## 2     1 0.05149690
```

```
sim_data = data.frame(y, X)
fit_lasso = train(
  y ~ ., data = sim_data,
  method = "glmnet",
  trControl = cv_5,
  tuneGrid = lasso_grid
)
fit_lasso$results
```

```
##   alpha     lambda  Accuracy     Kappa AccuracySD    KappaSD
## 1     1 0.03087158 0.7679304 0.5358028 0.03430230 0.06844656
## 2     1 0.05149690 0.7689003 0.5377583 0.02806941 0.05596114
```

24.5 External Links

- `glmnet` Web Vingette - Details from the package developers.

24.6 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "methods"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [7] "base"
```

- Additional Packages, Attached

```
## [1] "caret"   "ggplot2" "lattice" "broom"   "glmnet"  "foreach" "Matrix"
```

- Additional Packages, Not Attached

```
##  [1] "Rcpp"         "lubridate"    "tidyr"        "class"
##  [5] "assertthat"   "rprojroot"    "digest"       "ipred"
##  [9] "psych"        "R6"           "plyr"         "backports"
## [13] "stats4"       "evaluate"     "e1071"        "rlang"
## [17] "lazyeval"     "kernlab"      "rpart"        "rmarkdown"
## [21] "splines"      "CVST"         "ddalpha"      "gower"
## [25] "stringr"      "foreign"      "munsell"      "compiler"
## [29] "pkgconfig"    "mnormt"       "dimRed"       "htmltools"
## [33] "nnet"         "tibble"       "prodlim"      "DRR"
## [37] "bookdown"     "codetools"    "RcppRoll"     "dplyr"
## [41] "withr"        "MASS"         "recipes"      "ModelMetrics"
## [45] "grid"         "nlme"         "gtable"       "magrittr"
```

```
## [49] "scales"       "stringi"      "reshape2"     "bindrcpp"
## [53] "timeDate"     "robustbase"   "lava"         "iterators"
## [57] "tools"        "glue"         "DEoptimR"     "purrr"
## [61] "parallel"     "survival"     "yaml"         "colorspace"
## [65] "knitr"        "bindr"
```

Chapter 25

Elastic Net

We again use the `Hitters` dataset from the `ISLR` package to explore another shrinkage method, **elastic net**, which combines the *ridge* and *lasso* methods from the previous chapter.

25.1 Hitters Data

```
data(Hitters, package = "ISLR")
Hitters = na.omit(Hitters)
```

We again remove the missing data, which was all in the response variable, `Salary`.

```
tibble::as_tibble(Hitters)
```

```
## # A tibble: 263 x 20
##    AtBat  Hits HmRun  Runs   RBI Walks Years CAtBat CHits CHmRun CRuns
##  * <int> <int> <int> <int> <int> <int> <int>  <int> <int>  <int> <int>
##  1   315    81     7    24    38    39    14   3449   835     69   321
##  2   479   130    18    66    72    76     3   1624   457     63   224
##  3   496   141    20    65    78    37    11   5628  1575    225   828
##  4   321    87    10    39    42    30     2    396   101     12    48
##  5   594   169     4    74    51    35    11   4408  1133     19   501
##  6   185    37     1    23     8    21     2    214    42      1    30
##  7   298    73     0    24    24     7     3    509   108      0    41
##  8   323    81     6    26    32     8     2    341    86      6    32
##  9   401    92    17    49    66    65    13   5206  1332    253   784
## 10   574   159    21   107    75    59    10   4631  1300     90   702
## # ... with 253 more rows, and 9 more variables: CRBI <int>, CWalks <int>,
## #   League <fctr>, Division <fctr>, PutOuts <int>, Assists <int>,
## #   Errors <int>, Salary <dbl>, NewLeague <fctr>
```

```
dim(Hitters)
```

```
## [1] 263  20
```

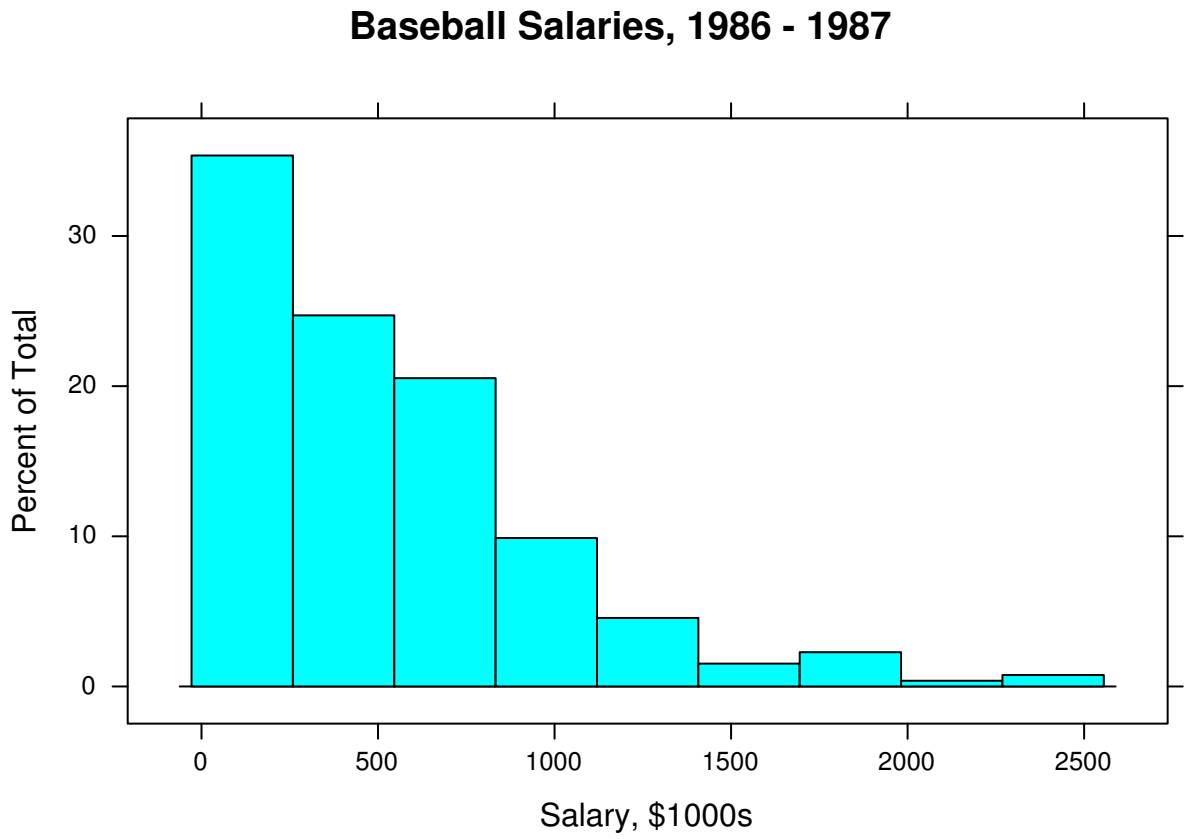
Because this dataset isn't particularly large, we will forego a test-train split, and simply use all of the data as training data.

```
# this is a temporary workaround for an issue with glmnet, Matrix, and R version 3.3.3
# see here: http://stackoverflow.com/questions/43282720/r-error-in-validobject-object-when-running-as-s
library(methods)
```

```
library(caret)
library(glmnet)
```

Since he have loaded `caret`, we also have access to the `lattice` package which has a nice histogram function.

```
histogram(Hitters$Salary, xlab = "Salary, $1000s", main = "Baseball Salaries, 1986 - 1987")
```



25.2 Elastic Net for Regression

Like ridge and lasso, we again attempt to minimize the residual sum of squares plus some penalty term.

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\left[(1-\alpha)||\beta||_2^2/2 + \alpha||\beta||_1\right]$$

Here, $||\beta||_1$ is called the l_1 norm.

$$||\beta||_1 = \sum_{j=1}^{p} |\beta_j|$$

Similarly, $||\beta||_2$ is called the l_2, or Euclidean norm.

$$||\beta||_2 = \sqrt{\sum_{j=1}^{p} \beta_j^2}$$

These both quantify how "large" the coefficients are. Like lasso and ridge, the intercept is not penalized and `glment` takes care of standardization internally. Also reported coefficients are on the original scale.

The new penalty is $\frac{\lambda \cdot (1-\alpha)}{2}$ times the ridge penalty plus $\lambda \cdot \alpha$ times the lasso lasso penalty. (Dividing the ridge penalty by 2 is a mathematical convenience for optimization.) Essentially, with the correct choice of λ and α these two "penalty coefficients" can be any positive numbers.

Often it is more useful to simply think of α as controlling the mixing between the two penalties and λ controlling the amount of penalization. α takes values between 0 and 1. Using $\alpha = 1$ gives the lasso that we have seen before. Similarly, $\alpha = 0$ gives ridge. We used these two before with `glmnet()` to specify which to method we wanted. Now we also allow for α values in between.

```
set.seed(430)
cv_5 = trainControl(method = "cv", number = 5)
```

We first setup our cross-validation strategy, which will be 5 fold. We then use `train()` with `method = "glmnet"` which is actually fitting the elastic net.

```
hit_elnet = train(
  Salary ~ ., data = Hitters,
  method = "glmnet",
  trControl = cv_5
)
```

First, note that since we are using `caret()` directly, it is taking care of dummy variable creation. So unlike before when we used `glmnet()`, we do not need to manually create a model matrix.

Also note that we have allowed `caret` to choose the tuning parameters for us.

```
hit_elnet
```

```
## glmnet
##
## 263 samples
##  19 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 210, 211, 210, 211, 210
## Resampling results across tuning parameters:
##
##   alpha  lambda   RMSE   Rsquared  MAE
##   0.10    0.5106  340.7  0.4463    242.4
##   0.10    5.1056  340.5  0.4441    241.2
```

```
##   0.10   51.0564  344.7  0.4329    237.3
##   0.55    0.5106  340.3  0.4468    242.1
##   0.55    5.1056  341.2  0.4426    239.3
##   0.55   51.0564  346.0  0.4345    238.6
##   1.00    0.5106  339.9  0.4475    241.7
##   1.00    5.1056  342.2  0.4400    237.9
##   1.00   51.0564  355.4  0.4162    246.3
##
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were alpha = 1 and lambda = 0.5106.
```

Notice a few things with these results. First, we have tried three α values, `0.1`, `0.55`, and `1`. It is not entirely clear why `caret` doesn't use `0`. It likely uses `0.1` to fit a model close to ridge, but with some potential for sparsity.

Here, the best result uses $\alpha = 0.55$, so this result is somewhere between ridge and lasso.

```
hit_elnet_int = train(
  Salary ~ . ^ 2, data = Hitters,
  method = "glmnet",
  trControl = cv_5,
  tuneLength = 10
)
```

Now we try a much larger model search. First, we're expanding the feature space to include all interactions. Since we are using penalized regression, we don't have to worry as much about overfitting. If many of the added variables are not useful, we will likely use a model close to lasso which makes many of them 0.

We're also using a larger tuning grid. By setting `tuneLength = 10`, we will search 10 α values and 10 λ values for each. Because of this larger tuning grid, the results will be very large.

To deal with this, we write a quick helper function to extract the row with the best tuning parameters.

```
get_best_result = function(caret_fit) {
  best_result = caret_fit$results[as.numeric(rownames(caret_fit$bestTune)), ]
  rownames(best_result) = NULL
  best_result
}
```

We then call this function on the trained object.

```
get_best_result(hit_elnet_int)
```

```
##   alpha lambda  RMSE Rsquared   MAE RMSESD RsquaredSD MAESD
## 1     1  4.135 295.5   0.5808 196.8     50     0.1525 11.49
```

We see that the best result uses $\alpha = 1$, which makes since. With $\alpha = 1$, many of the added interaction coefficients are likely set to zero. (Unfortunately, obtaining this information after using `caret` with `glmnet` isn't easy. The two don't actually play very nice together. We'll use `cv.glmnet()` with the expanded feature space to explore this.)

Also, this CV-RMSE is better than the lasso and ridge from the previous chapter that did not use the expanded feature space.

We also perform a quick analysis using `cv.glmnet()` instead. Due in part to randomness in cross validation, and differences in how `cv.glmnet()` and `train()` search for λ, the results are slightly different.

```
set.seed(430)
X = model.matrix(Salary ~ . ^ 2, Hitters)[, -1]
y = Hitters$Salary

fit_lasso_cv = cv.glmnet(X, y, alpha = 1)
sqrt(fit_lasso_cv$cvm[fit_lasso_cv$lambda == fit_lasso_cv$lambda.min]) # CV-RMSE minimum
```

```
## [1] 304.1
```

The commented line is not run, since it produces a lot of output, but if run, it will show that the fast majority of the coefficients are zero! (Also, you'll notice that `cv.glmnet()` does not respect the usual predictor hierarchy. Not a problem for prediction, but a massive interpretation issue!)

```
#coef(fit_lasso_cv)
```

25.3 Elastic Net for Classification

Above, we have performed a regression task. But like lasso and ridge, elastic net can also be used for classification by using the deviance instead of the residual sum of squares. This essentially happens automatically in `caret` if the response variable is a factor.

We'll test this using the familiar `Default` dataset, which we first test-train split.

```
data(Default, package = "ISLR")
```

```
set.seed(430)
default_idx = createDataPartition(Default$default, p = 0.75, list = FALSE)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

We then fit an elastic net with a default tuning grid.

```
def_elnet = train(
  default ~ ., data = default_trn,
  method = "glmnet",
  trControl = cv_5
)
def_elnet
```

```
## glmnet
##
## 7501 samples
##    3 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 6001, 6001, 6000, 6001, 6001
## Resampling results across tuning parameters:
##
```

```
##   alpha  lambda     Accuracy  Kappa
##   0.10   0.0001242  0.9733    0.40751
##   0.10   0.0012424  0.9727    0.36354
##   0.10   0.0124239  0.9676    0.07718
##   0.55   0.0001242  0.9732    0.40600
##   0.55   0.0012424  0.9724    0.36841
##   0.55   0.0124239  0.9684    0.12506
##   1.00   0.0001242  0.9732    0.40600
##   1.00   0.0012424  0.9727    0.38095
##   1.00   0.0124239  0.9689    0.15170
##
## Accuracy was used to select the optimal model using  the largest value.
## The final values used for the model were alpha = 0.1 and lambda
##  = 0.0001242.
```

Since the best model used $\alpha = 1$, this is a lasso model.

We also try an expanded feature space, and a larger tuning grid.

```
def_elnet_int = train(
  default ~ . ^ 2, data = default_trn,
  method = "glmnet",
  trControl = cv_5,
  tuneLength = 10
)
```

Since the result here will return 100 models, we again use are helper function to simply extract the best result.

```
get_best_result(def_elnet_int)
```

```
##   alpha   lambda Accuracy  Kappa AccuracySD KappaSD
## 1     1 0.001888   0.9728 0.3906   0.002509 0.07252
```

Here we see $\alpha = 0.3$, which is a mix between ridge and lasso.

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

Evaluating the test accuracy of this model, we obtain one of the highest accuracies for this dataset of all methods we have tried.

```
# test acc
accuracy(actual = default_tst$default,
         predicted = predict(def_elnet_int, newdata = default_tst))
```

```
## [1] 0.9744
```

25.4 External Links

- `glmnet` Web Vingette - Details from the package developers.
- `glmnet` with `caret` - Some details on Elastic Net tuning in the `caret` package.

25.5 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "methods"   "stats"     "graphics"  "grDevices" "utils"     "datasets"
## [7] "base"
```

- Additional Packages, Attached

```
## [1] "glmnet"  "foreach" "Matrix"  "caret"   "ggplot2" "lattice"
```

- Additional Packages, Not Attached

```
##  [1] "reshape2"     "kernlab"      "purrr"        "splines"
##  [5] "colorspace"   "htmltools"    "stats4"       "yaml"
##  [9] "survival"     "prodlim"      "rlang"        "e1071"
## [13] "ModelMetrics" "withr"        "glue"         "bindrcpp"
## [17] "plyr"         "bindr"        "dimRed"       "lava"
## [21] "robustbase"   "stringr"      "timeDate"     "munsell"
## [25] "gtable"       "recipes"      "codetools"    "evaluate"
## [29] "knitr"        "class"        "DEoptimR"     "Rcpp"
## [33] "scales"       "backports"    "ipred"        "CVST"
## [37] "digest"       "stringi"      "bookdown"     "dplyr"
## [41] "RcppRoll"     "ddalpha"      "grid"         "rprojroot"
## [45] "tools"        "magrittr"     "lazyeval"     "tibble"
## [49] "DRR"          "pkgconfig"    "MASS"         "lubridate"
## [53] "gower"        "assertthat"   "rmarkdown"    "iterators"
## [57] "R6"           "rpart"        "nnet"         "nlme"
## [61] "compiler"
```

Chapter 26

Trees

```
library(tree)
```

In this document, we will use the package `tree` for both classification and regression trees. Note that there are many packages to do this in `R`. `rpart` may be the most common, however, we will use `tree` for simplicity.

26.1 Classification Trees

```
library(ISLR)
```

To understand classification trees, we will use the `Carseat` dataset from the `ISLR` package. We will first modify the response variable `Sales` from its original use as a numerical variable, to a categorical variable with `High` for high sales, and `Low` for low sales.

```
data(Carseats)
#?Carseats
str(Carseats)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ Sales      : num  9.5 11.22 10.06 7.4 4.15 ...
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
```

```
Carseats$Sales = as.factor(ifelse(Carseats$Sales <= 8, "Low", "High"))
str(Carseats)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ Sales      : Factor w/ 2 levels "High","Low": 1 1 1 2 2 1 2 1 2 2 ...
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
```

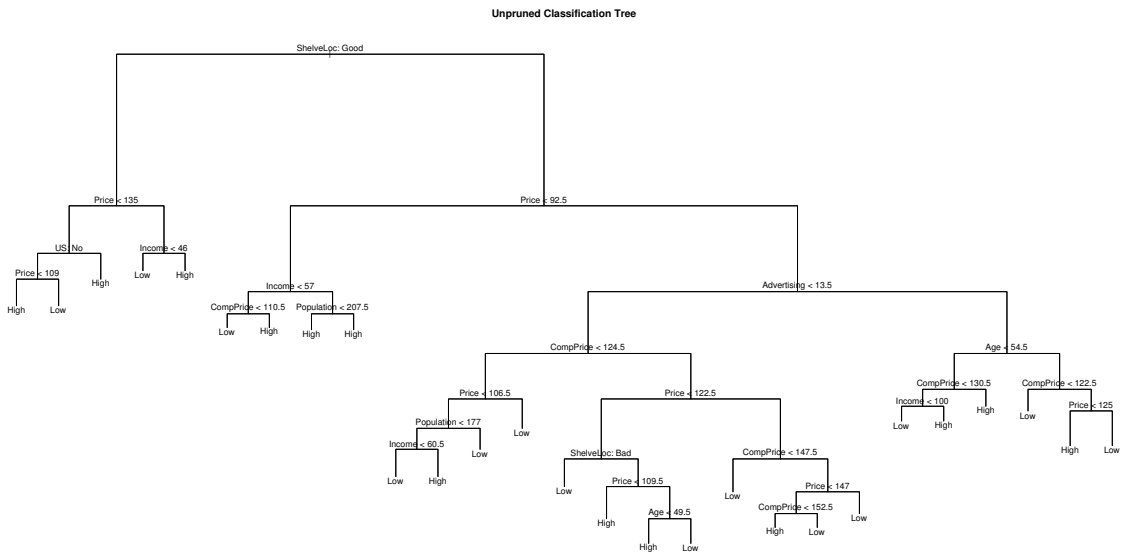
We first fit an unpruned classification tree using all of the predictors. Details of this process can be found using `?tree` and `?tree.control`

```
seat_tree = tree(Sales ~ ., data = Carseats)
# seat_tree = tree(Sales ~ ., data = Carseats,
#                  control = tree.control(nobs = nrow(Carseats), minsize = 10))
summary(seat_tree)
```

```
##
## Classification tree:
## tree(formula = Sales ~ ., data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "US"          "Income"      "CompPrice"
## [6] "Population"  "Advertising" "Age"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

We see this tree has 27 terminal nodes and a misclassification rate of 0.09.

```
plot(seat_tree)
text(seat_tree, pretty = 0)
title(main = "Unpruned Classification Tree")
```



Above we plot the tree. Below we output the details of the splits.

```
seat_tree
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 Low ( 0.41000 0.59000 )
##     2) ShelveLoc: Good 85  90.330 High ( 0.77647 0.22353 )
##       4) Price < 135 68  49.260 High ( 0.88235 0.11765 )
##         8) US: No 17  22.070 High ( 0.64706 0.35294 )
##          16) Price < 109 8   0.000 High ( 1.00000 0.00000 ) *
##          17) Price > 109 9  11.460 Low ( 0.33333 0.66667 ) *
##         9) US: Yes 51  16.880 High ( 0.96078 0.03922 ) *
##       5) Price > 135 17  22.070 Low ( 0.35294 0.64706 )
##        10) Income < 46 6   0.000 Low ( 0.00000 1.00000 ) *
##        11) Income > 46 11  15.160 High ( 0.54545 0.45455 ) *
##     3) ShelveLoc: Bad,Medium 315 390.600 Low ( 0.31111 0.68889 )
##       6) Price < 92.5 46  56.530 High ( 0.69565 0.30435 )
##        12) Income < 57 10  12.220 Low ( 0.30000 0.70000 )
##          24) CompPrice < 110.5 5   0.000 Low ( 0.00000 1.00000 ) *
##          25) CompPrice > 110.5 5   6.730 High ( 0.60000 0.40000 ) *
##        13) Income > 57 36  35.470 High ( 0.80556 0.19444 )
##          26) Population < 207.5 16  21.170 High ( 0.62500 0.37500 ) *
##          27) Population > 207.5 20   7.941 High ( 0.95000 0.05000 ) *
##       7) Price > 92.5 269 299.800 Low ( 0.24535 0.75465 )
##        14) Advertising < 13.5 224 213.200 Low ( 0.18304 0.81696 )
##          28) CompPrice < 124.5 96  44.890 Low ( 0.06250 0.93750 )
##            56) Price < 106.5 38  33.150 Low ( 0.15789 0.84211 )
##             112) Population < 177 12  16.300 Low ( 0.41667 0.58333 )
##               224) Income < 60.5 6   0.000 Low ( 0.00000 1.00000 ) *
##               225) Income > 60.5 6   5.407 High ( 0.83333 0.16667 ) *
##             113) Population > 177 26   8.477 Low ( 0.03846 0.96154 ) *
##            57) Price > 106.5 58   0.000 Low ( 0.00000 1.00000 ) *
```

```
##           29) CompPrice > 124.5 128 150.200 Low ( 0.27344 0.72656 )
##             58) Price < 122.5 51  70.680 High ( 0.50980 0.49020 )
##              116) ShelveLoc: Bad 11   6.702 Low ( 0.09091 0.90909 ) *
##              117) ShelveLoc: Medium 40  52.930 High ( 0.62500 0.37500 )
##                234) Price < 109.5 16   7.481 High ( 0.93750 0.06250 ) *
##                235) Price > 109.5 24  32.600 Low ( 0.41667 0.58333 )
##                  470) Age < 49.5 13  16.050 High ( 0.69231 0.30769 ) *
##                  471) Age > 49.5 11   6.702 Low ( 0.09091 0.90909 ) *
##             59) Price > 122.5 77  55.540 Low ( 0.11688 0.88312 )
##              118) CompPrice < 147.5 58  17.400 Low ( 0.03448 0.96552 ) *
##              119) CompPrice > 147.5 19  25.010 Low ( 0.36842 0.63158 )
##                238) Price < 147 12  16.300 High ( 0.58333 0.41667 )
##                  476) CompPrice < 152.5 7   5.742 High ( 0.85714 0.14286 ) *
##                  477) CompPrice > 152.5 5   5.004 Low ( 0.20000 0.80000 ) *
##                239) Price > 147 7   0.000 Low ( 0.00000 1.00000 ) *
##         15) Advertising > 13.5 45  61.830 High ( 0.55556 0.44444 )
##           30) Age < 54.5 25  25.020 High ( 0.80000 0.20000 )
##             60) CompPrice < 130.5 14  18.250 High ( 0.64286 0.35714 )
##              120) Income < 100 9  12.370 Low ( 0.44444 0.55556 ) *
##              121) Income > 100 5   0.000 High ( 1.00000 0.00000 ) *
##             61) CompPrice > 130.5 11   0.000 High ( 1.00000 0.00000 ) *
##           31) Age > 54.5 20  22.490 Low ( 0.25000 0.75000 )
##             62) CompPrice < 122.5 10   0.000 Low ( 0.00000 1.00000 ) *
##             63) CompPrice > 122.5 10  13.860 Low ( 0.50000 0.50000 )
##              126) Price < 125 5   0.000 High ( 1.00000 0.00000 ) *
##              127) Price > 125 5   0.000 Low ( 0.00000 1.00000 ) *
```

We now test-train split the data so we can evaluate how well our tree is working. We use 200 observations for each.

```
dim(Carseats)
```

```
## [1] 400  11
```

```
set.seed(2)
seat_idx = sample(1:nrow(Carseats), 200)
seat_trn = Carseats[seat_idx,]
seat_tst = Carseats[-seat_idx,]
```

```
seat_tree = tree(Sales ~ ., data = seat_trn)
```

```
summary(seat_tree)
```

```
##
## Classification tree:
## tree(formula = Sales ~ ., data = seat_trn)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Population"  "Advertising" "Income"
## [6] "Age"         "CompPrice"
## Number of terminal nodes:  19
## Residual mean deviance:  0.4282 = 77.51 / 181
## Misclassification error rate: 0.105 = 21 / 200
```

Note that, the tree is not using all of the available variables.

```
summary(seat_tree)$used
```

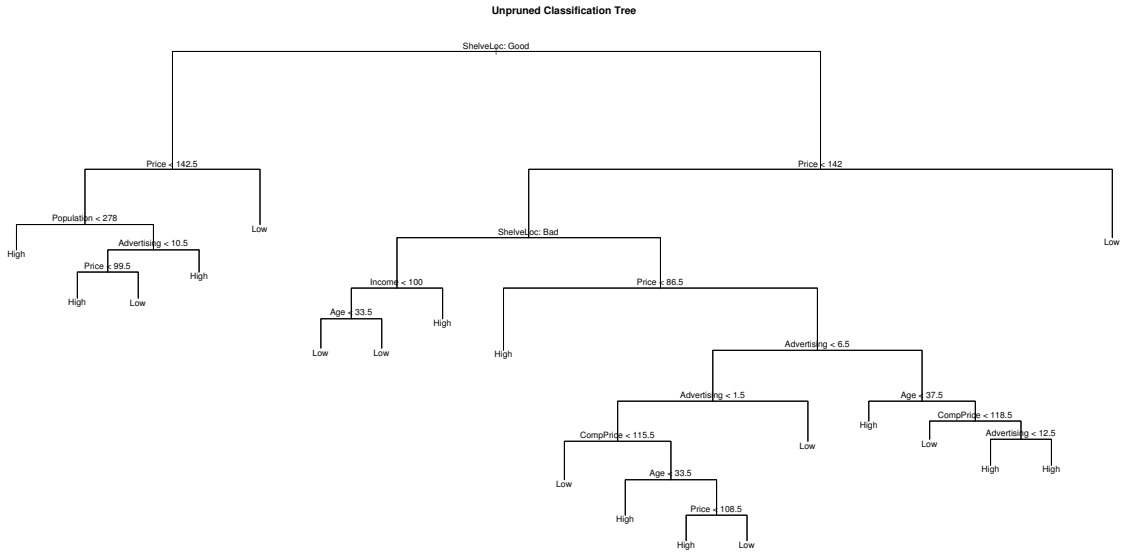
```
## [1] ShelveLoc   Price       Population  Advertising Income      Age
## [7] CompPrice
## 11 Levels: <leaf> CompPrice Income Advertising Population ... US
```

```
names(Carseats)[which(!(names(Carseats) %in% summary(seat_tree)$used))]
```

```
## [1] "Sales"     "Education" "Urban"     "US"
```

Also notice that, this new tree is slightly different than the tree fit to all of the data.

```
plot(seat_tree)
text(seat_tree, pretty = 0)
title(main = "Unpruned Classification Tree")
```



When using the `predict()` function on a tree, the default `type` is `vector` which gives predicted probabilities for both classes. We will use `type = class` to directly obtain classes. We first fit the tree using the training data (above), then obtain predictions on both the train and test set, then view the confusion matrix for both.

```
seat_trn_pred = predict(seat_tree, seat_trn, type = "class")
seat_tst_pred = predict(seat_tree, seat_tst, type = "class")
#predict(seat_tree, seat_trn, type = "vector")
#predict(seat_tree, seat_tst, type = "vector")
```

```
# train confusion
table(predicted = seat_trn_pred, actual = seat_trn$Sales)
```

```
##           actual
## predicted High Low
##      High   66  10
##      Low    14 110
```

```
# test confusion
table(predicted = seat_tst_pred, actual = seat_tst$Sales)
```

```
##           actual
## predicted High Low
##      High   57  29
##      Low    27  87
```

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

```
# train acc
accuracy(predicted = seat_trn_pred, actual = seat_trn$Sales)
```

```
## [1] 0.88
```

```
# test acc
accuracy(predicted = seat_tst_pred, actual = seat_tst$Sales)
```

```
## [1] 0.72
```

Here it is easy to see that the tree has been over-fit. The train set performs much better than the test set.

We will now use cross-validation to find a tree by considering trees of different sizes which have been pruned from our original tree.

```
set.seed(3)
seat_tree_cv = cv.tree(seat_tree, FUN = prune.misclass)
```

```
# index of tree with minimum error
min_idx = which.min(seat_tree_cv$dev)
min_idx
```

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

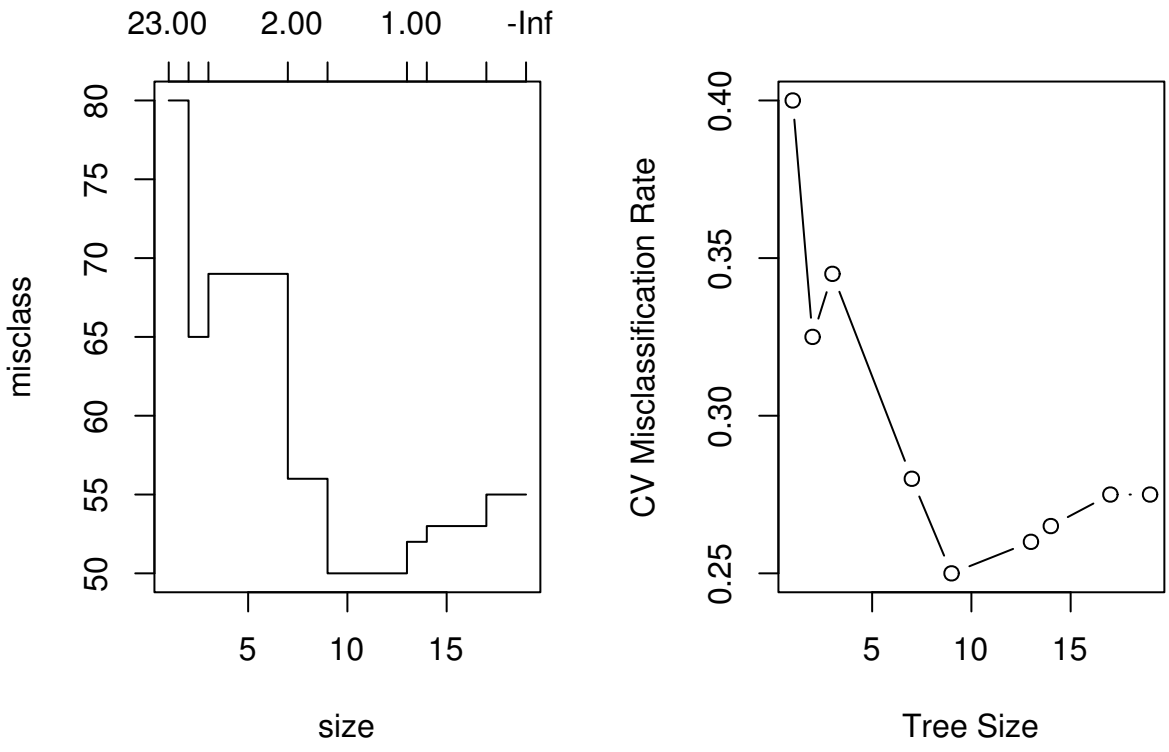
```
# number of terminal nodes in that tree
seat_tree_cv$size[min_idx]
```

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

```
# misclassification rate of each tree
seat_tree_cv$dev / length(seat_idx)
```

```
## [1] 0.275 0.275 0.265 0.260 0.250 0.280 0.345 0.325 0.400
```

```
par(mfrow = c(1, 2))
# default plot
plot(seat_tree_cv)
# better plot
plot(seat_tree_cv$size, seat_tree_cv$dev / nrow(seat_trn), type = "b",
     xlab = "Tree Size", ylab = "CV Misclassification Rate")
```

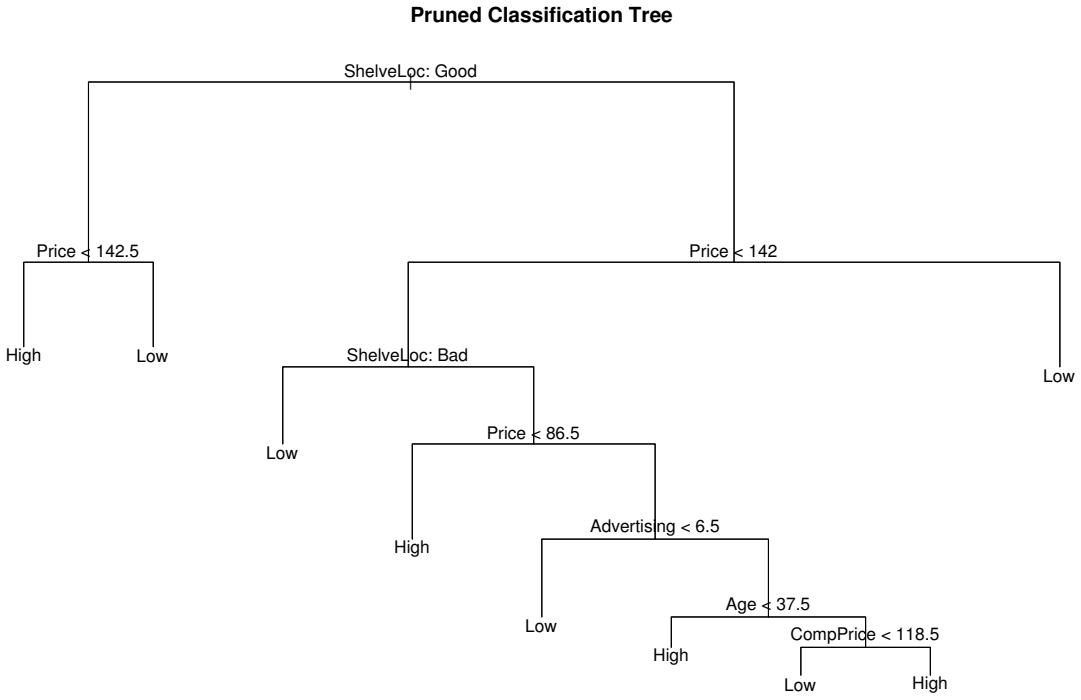


It appears that a tree of size 9 has the fewest misclassifications of the considered trees, via cross-validation. We use `prune.misclass()` to obtain that tree from our original tree, and plot this smaller tree.

```
seat_tree_prune = prune.misclass(seat_tree, best = 9)
summary(seat_tree_prune)
```

```
##
## Classification tree:
## snip.tree(tree = seat_tree, nodes = c(223L, 4L, 12L, 54L))
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Advertising" "Age"         "CompPrice"
## Number of terminal nodes:  9
## Residual mean deviance:  0.8103 = 154.8 / 191
## Misclassification error rate: 0.155 = 31 / 200
```

```
plot(seat_tree_prune)
text(seat_tree_prune, pretty = 0)
title(main = "Pruned Classification Tree")
```



We again obtain predictions using this smaller tree, and evaluate on the test and train sets.

```
# train
seat_prune_trn_pred = predict(seat_tree_prune, seat_trn, type = "class")
table(predicted = seat_prune_trn_pred, actual = seat_trn$Sales)
```

```
##          actual
## predicted High Low
##      High   59  10
##      Low    21 110
```

```
accuracy(predicted = seat_prune_trn_pred, actual = seat_trn$Sales)
```

```
## [1] 0.845
```

```
# test
seat_prune_tst_pred = predict(seat_tree_prune, seat_tst, type = "class")
table(predicted = seat_prune_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   60  22
##      Low    24  94
```

```
accuracy(predicted = seat_prune_tst_pred, actual = seat_tst$Sales)
```

```
## [1] 0.77
```

The train set has performed almost as well as before, and there was a **small** improvement in the test set, but it is still obvious that we have over-fit. Trees tend to do this. We will look at several ways to fix this, including: bagging, boosting and random forests.

26.2 Regression Trees

To demonstrate regression trees, we will use the `Boston` data. Recall `medv` is the response. We first split the data in half.

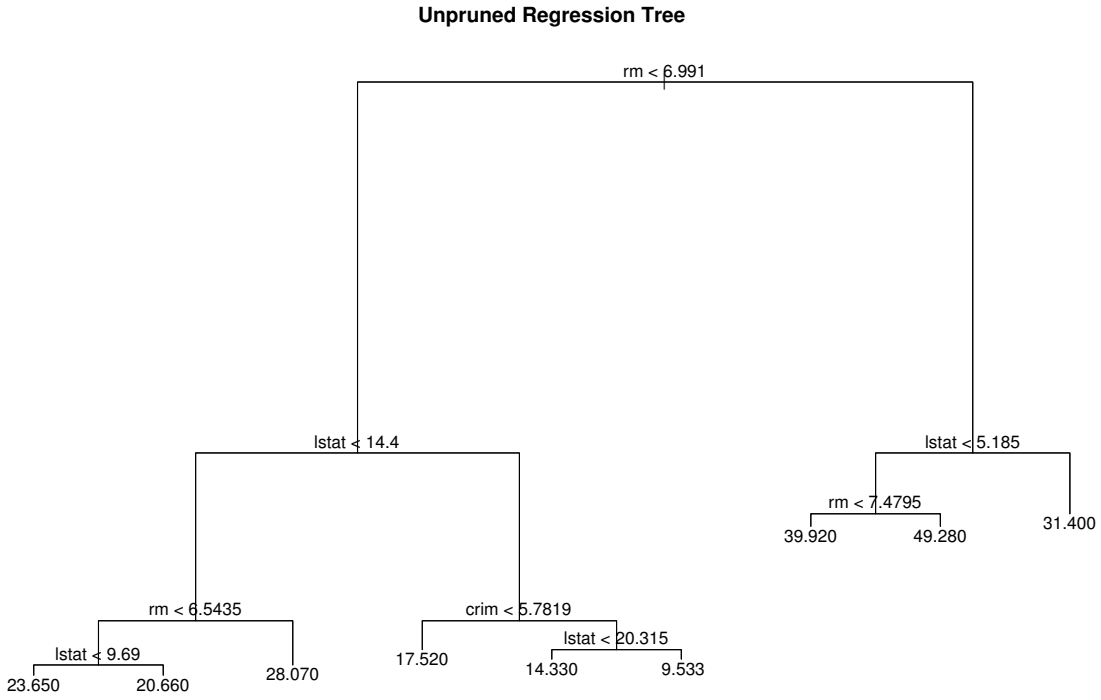
```
library(MASS)
set.seed(18)
boston_idx = sample(1:nrow(Boston), nrow(Boston) / 2)
boston_trn = Boston[boston_idx,]
boston_tst = Boston[-boston_idx,]
```

Then fit an unpruned regression tree to the training data.

```
boston_tree = tree(medv ~ ., data = boston_trn)
summary(boston_tree)
```

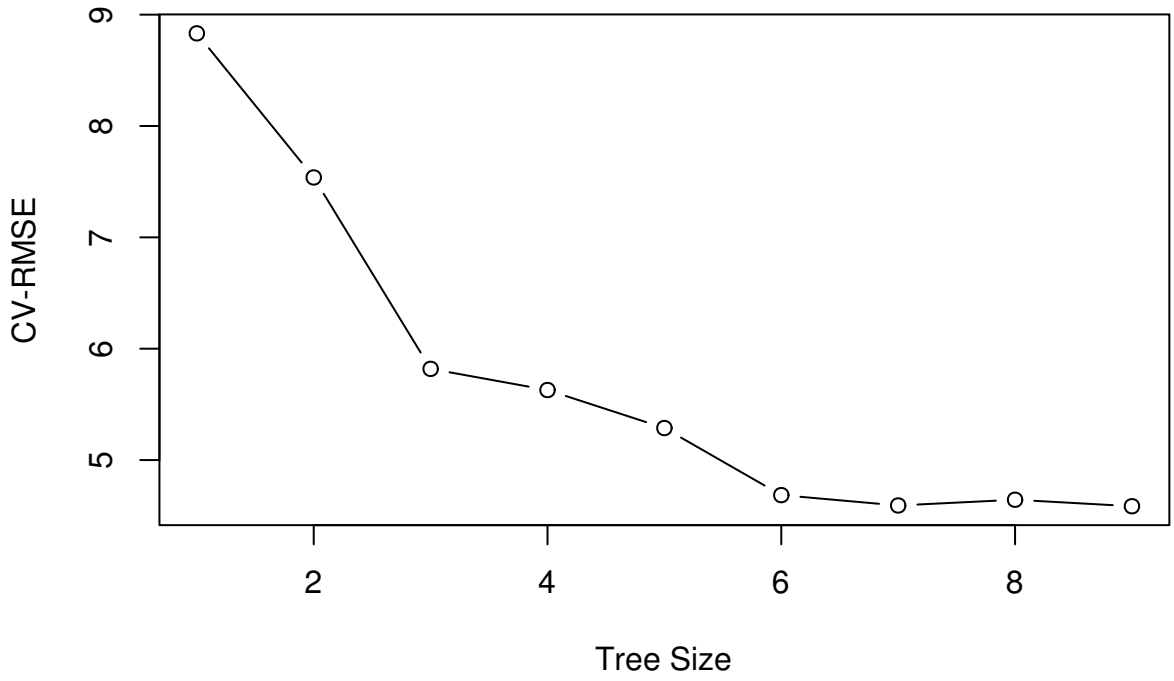
```
##
## Regression tree:
## tree(formula = medv ~ ., data = boston_trn)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"
## Number of terminal nodes:  9
## Residual mean deviance:  12.35 = 3013 / 244
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -13.600  -1.832  -0.120   0.000   1.348  26.350
```

```
plot(boston_tree)
text(boston_tree, pretty = 0)
title(main = "Unpruned Regression Tree")
```



As with classification trees, we can use cross-validation to select a good pruning of the tree.

```
set.seed(18)
boston_tree_cv = cv.tree(boston_tree)
plot(boston_tree_cv$size, sqrt(boston_tree_cv$dev / nrow(boston_trn)), type = "b",
     xlab = "Tree Size", ylab = "CV-RMSE")
```

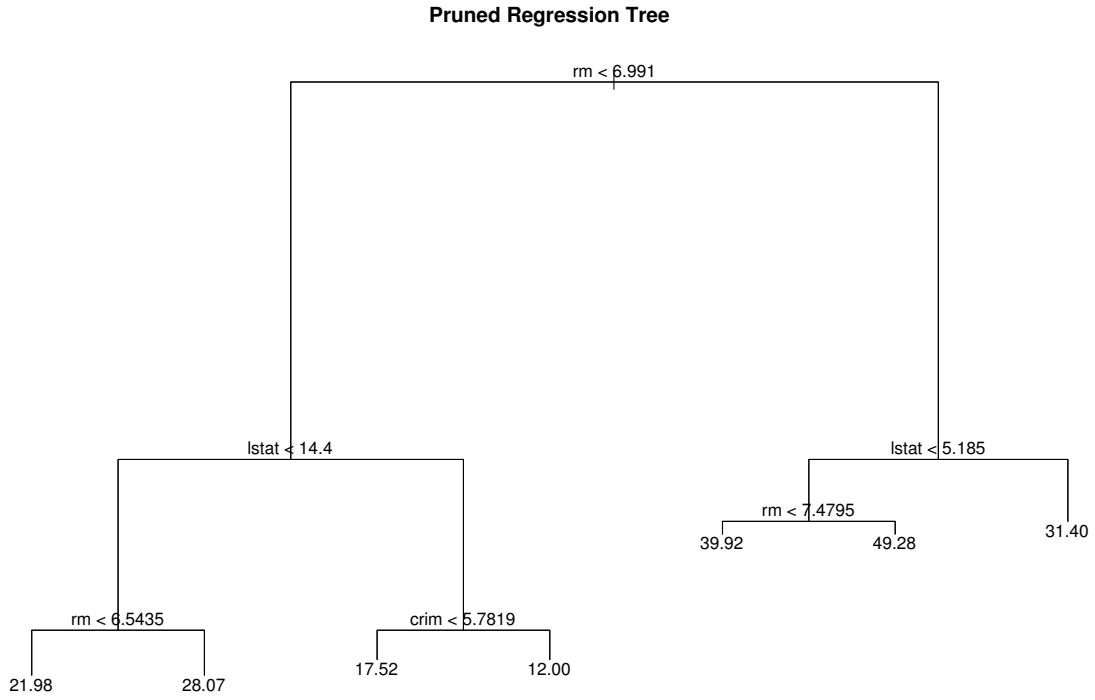


While the tree of size 9 does have the lowest RMSE, we'll prune to a size of 7 as it seems to perform just as well. (Otherwise we would not be pruning.) The pruned tree is, as expected, smaller and easier to interpret.

```
boston_tree_prune = prune.tree(boston_tree, best = 7)
summary(boston_tree_prune)
```

```
##
## Regression tree:
## snip.tree(tree = boston_tree, nodes = c(11L, 8L))
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"
## Number of terminal nodes:  7
## Residual mean deviance:  14.05 = 3455 / 246
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -13.60000  -2.12000   0.01731   0.00000   1.88000  28.02000
```

```
plot(boston_tree_prune)
text(boston_tree_prune, pretty = 0)
title(main = "Pruned Regression Tree")
```



Let's compare this regression tree to an additive linear model and use RMSE as our metric.

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

We obtain predictions on the train and test sets from the pruned tree. We also plot actual vs predicted. This plot may look odd. We'll compare it to a plot for linear regression below.

```
# training RMSE two ways
sqrt(summary(boston_tree_prune)$dev / nrow(boston_trn))
```

```
## [1] 3.695598
```

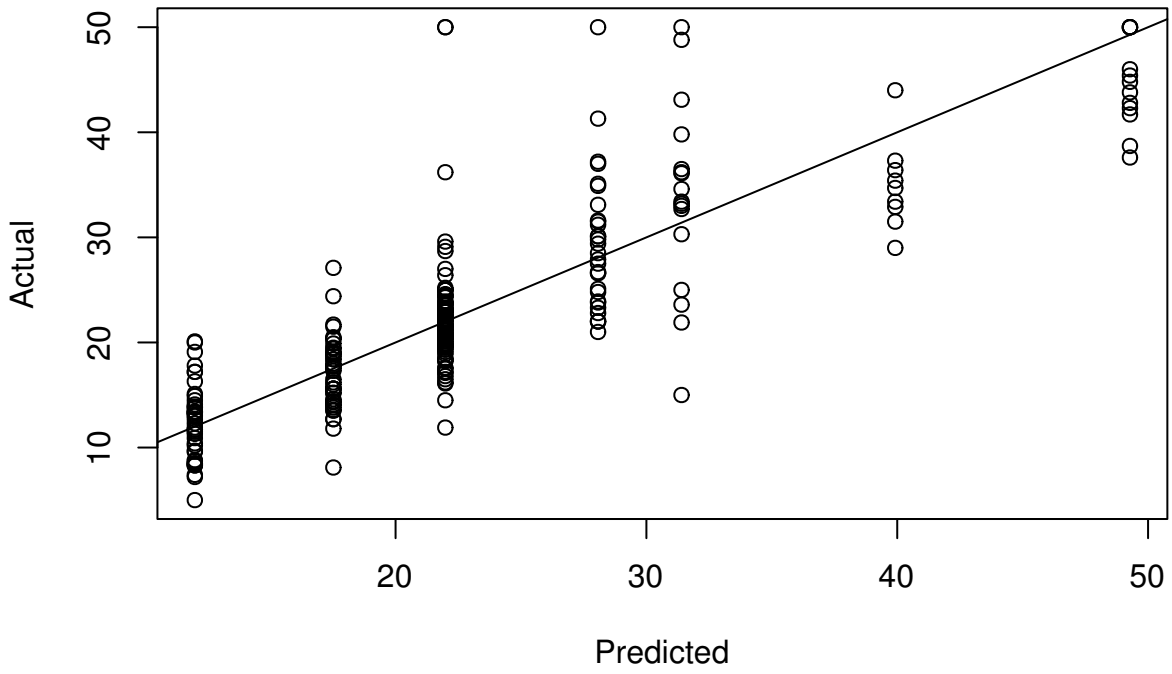
```
boston_prune_trn_pred = predict(boston_tree_prune, newdata = boston_trn)
rmse(boston_prune_trn_pred, boston_trn$medv)
```

```
## [1] 3.695598
```

```
# test RMSE
boston_prune_tst_pred = predict(boston_tree_prune, newdata = boston_tst)
rmse(boston_prune_tst_pred, boston_tst$medv)
```

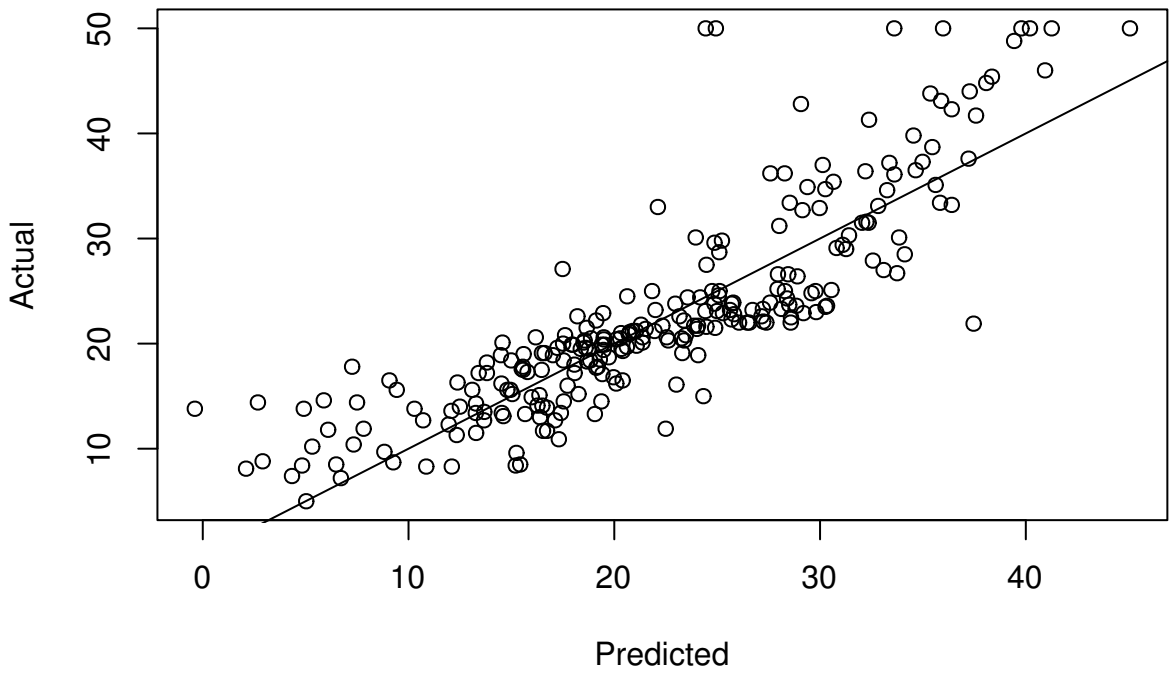
```
## [1] 5.331457
```

```
plot(boston_prune_tst_pred, boston_tst$medv, xlab = "Predicted", ylab = "Actual")
abline(0, 1)
```



Here, using an additive linear regression the actual vs predicted looks much more like what we are used to.

```
bostom_lm = lm(medv ~ ., data = boston_trn)
boston_lm_pred = predict(bostom_lm, newdata = boston_tst)
plot(boston_lm_pred, boston_tst$medv, xlab = "Predicted", ylab = "Actual")
abline(0, 1)
```




```
rmse(boston_lm_pred, boston_tst$medv)
```

```
## [1] 5.125877
```

We also see a lower test RMSE. The most obvious linear regression beats the tree! Again, we'll improve on this tree soon. Also note the summary of the additive linear regression below. Which is easier to interpret, that output, or the small tree above?

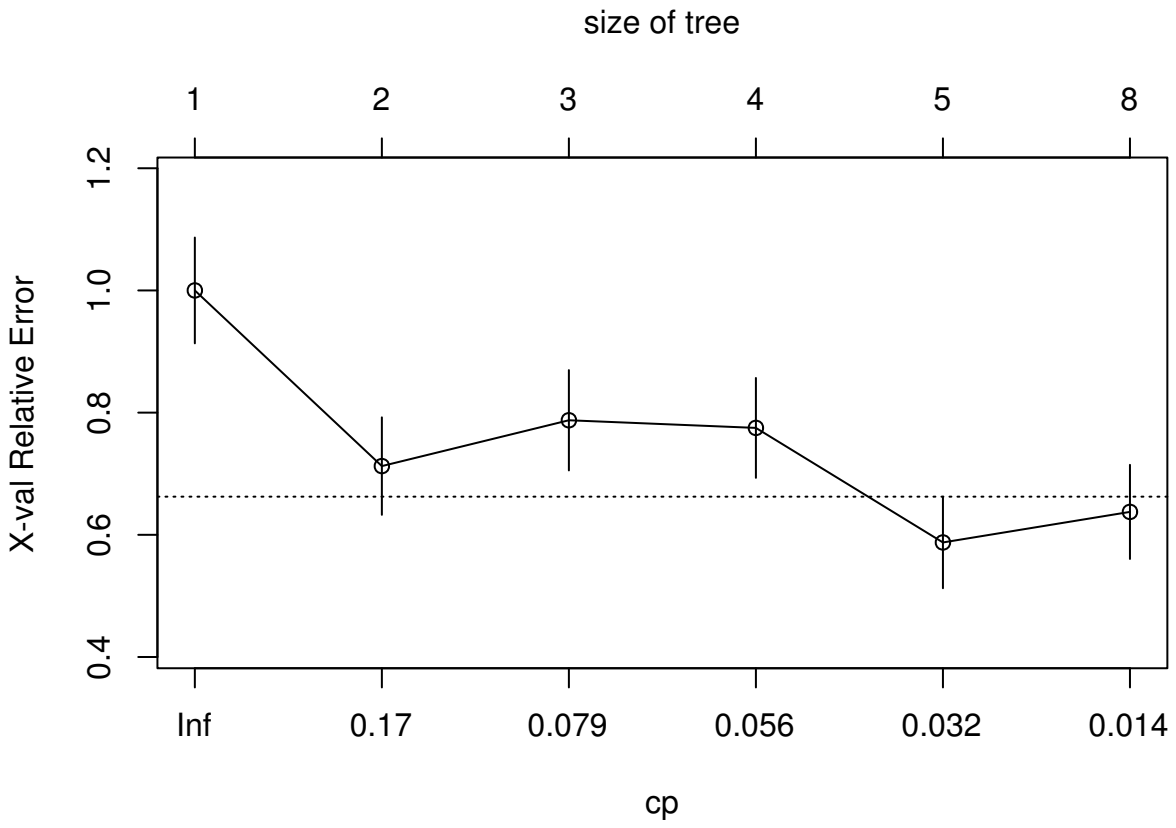
```
coef(bostom_lm)
```

```
##   (Intercept)          crim            zn         indus          chas
##  43.340158284  -0.113490889   0.046881038   0.018046856   3.557944155
##           nox            rm           age           dis           rad
## -21.904534125   3.486780787  -0.010592511  -1.766227892   0.354167931
##           tax       ptratio         black         lstat
##  -0.015036451  -0.830144898   0.003722857  -0.576134200
```

26.3 rpart Package

The `rpart` package is an alternative method for fitting trees in `R`. It is much more feature rich, including fitting multiple cost complexities and performing cross-validation by default. It also has the ability to produce much nicer trees. Based on its default settings, it will often result in smaller trees than using the `tree` package. See the references below for more information. `rpart` can also be tuned via `caret`.

```
library(rpart)
set.seed(430)
# Fit a decision tree using rpart
# Note: when you fit a tree using rpart, the fitting routine automatically
# performs 10-fold CV and stores the errors for later use
# (such as for pruning the tree)

# fit a tree using rpart
seat_rpart = rpart(Sales ~ ., data = seat_trn, method = "class")

# plot the cv error curve for the tree
# rpart tries different cost-complexities by default
# also stores cv results
plotcp(seat_rpart)
```

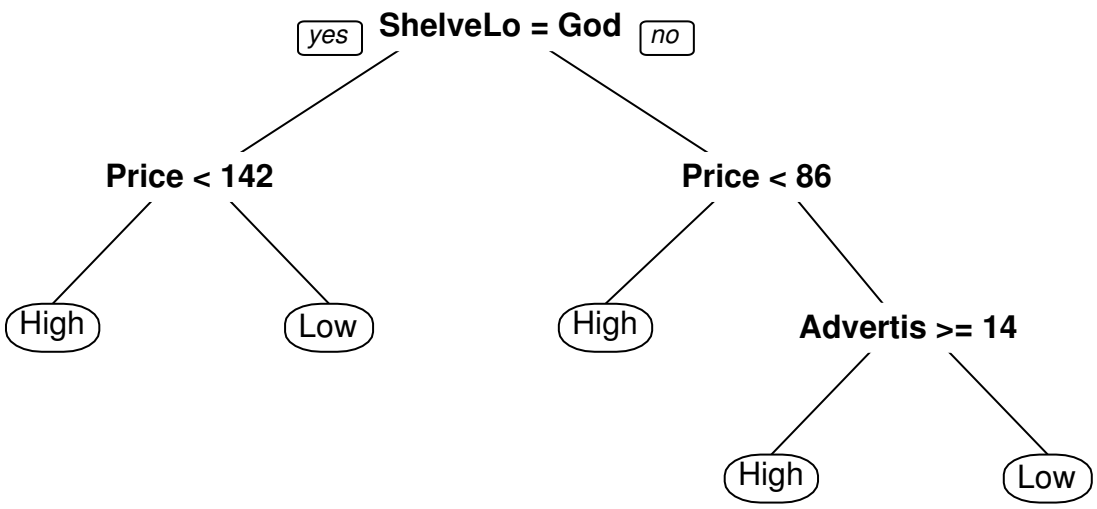



```
# find best value of cp
min_cp = seat_rpart$cptable[which.min(seat_rpart$cptable[,"xerror"]),"CP"]
min_cp
```

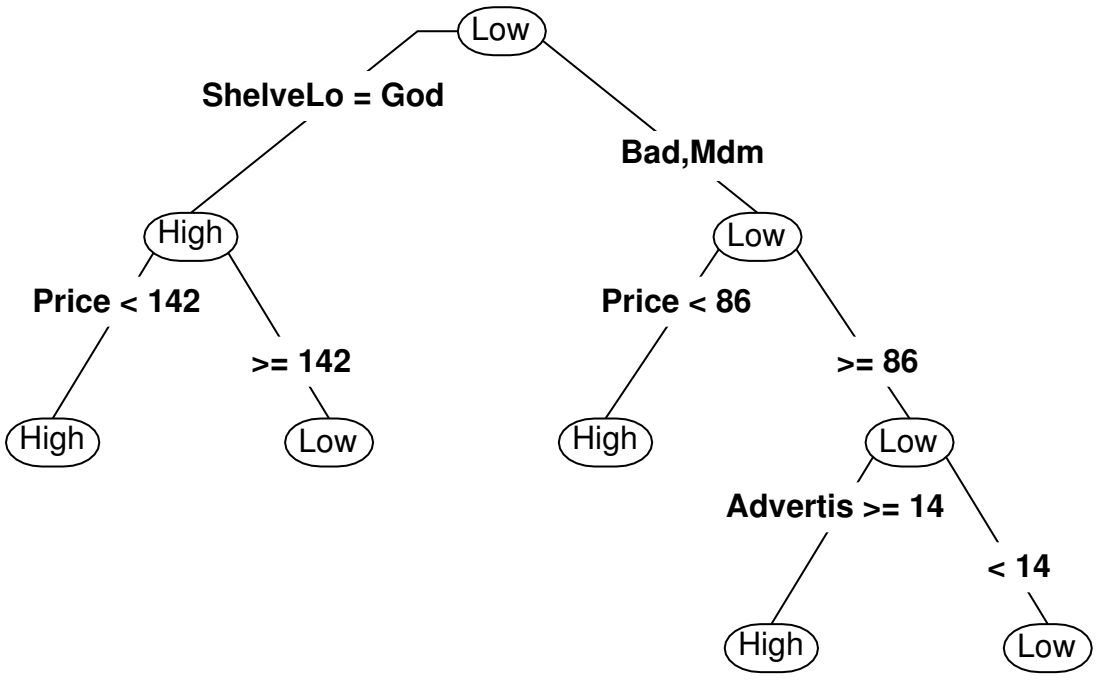
```
## [1] 0.02083333
```

```
# prunce tree using best cp
seat_rpart_prune = prune(seat_rpart, cp = min_cp)
```

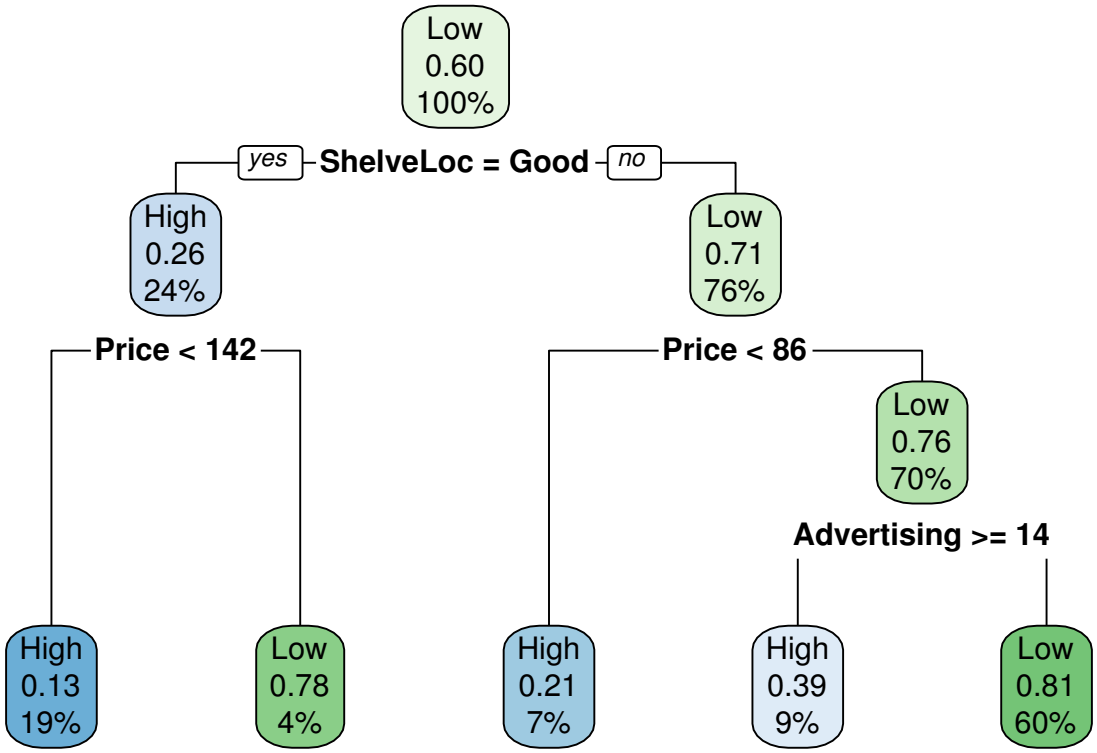
```
# nicer plots
library(rpart.plot)
prp(seat_rpart_prune)
```




```
prp(seat_rpart_prune, type = 4)
```




```
rpart.plot(seat_rpart_prune)
```



26.4 External Links

- An Introduction to Recursive Partitioning Using the `rpart` Routines - Details of the `rpart` package.
- `rpart.plot` Package - Detailed manual on plotting with `rpart` using the `rpart.plot` package.

26.5 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
## [1] "rpart.plot" "rpart"      "MASS"       "ISLR"       "tree"
```

- Additional Packages, Not Attached

```
##  [1] "Rcpp"      "bookdown"  "digest"    "rprojroot" "backports"
##  [6] "magrittr"  "evaluate"  "stringi"   "rmarkdown" "tools"
## [11] "stringr"   "yaml"      "compiler"  "htmltools" "knitr"
## [16] "methods"
```

Chapter 27

Ensemble Methods

We'll now consider ensembles of trees.

27.1 Regression

We first consider the regression case, using the `Boston` data from the `MASS` package. We will use RMSE as our metric, so we write a function which will help us along the way.

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

We also load all of the packages that we will need.

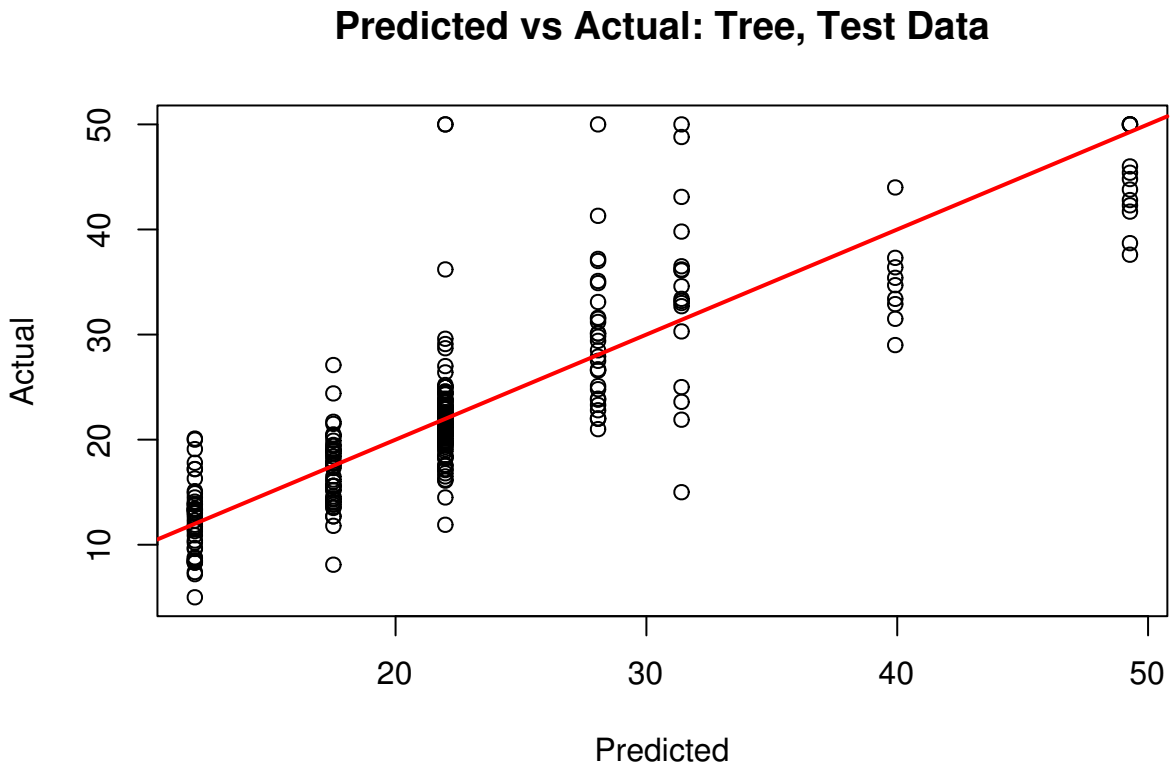
```
library(tree)
library(MASS)
library(ISLR)
library(randomForest)
library(gbm)
library(caret)
```

We first test-train split the data and fit the same pruned tree as before. (Note: When pruning the tree, the best tree is actually the unpruned tree. View the results of `cv.tree` to see this. However, we select the tree of size 7 as the best of the pruned trees.)

```
set.seed(18)
boston_idx = sample(1:nrow(Boston), nrow(Boston) / 2)
boston_trn = Boston[boston_idx,]
boston_tst = Boston[-boston_idx,]
```

27.1.1 Tree Model

```
boston_tree = tree(medv ~ ., data = boston_trn)
set.seed(18)
boston_tree_cv = cv.tree(boston_tree)
boston_tree_prune = prune.tree(boston_tree, best = 7)
boston_prune_tst_pred = predict(boston_tree_prune, newdata = boston_tst)
plot(boston_prune_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Tree, Test Data")
abline(0, 1, col = "red", lwd = 2)
```




```
(tree_tst_rmse = rmse(boston_prune_tst_pred, boston_tst$medv))
```

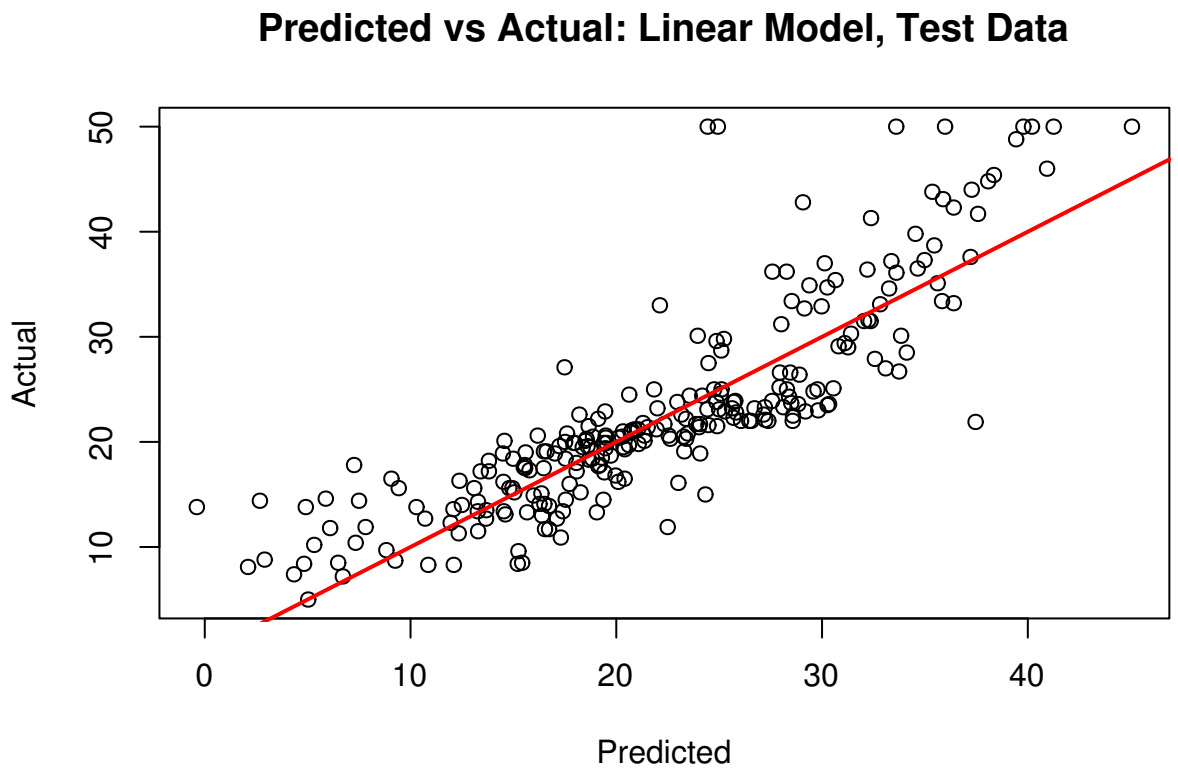
```
## [1] 5.331457
```

27.1.2 Linear Model

Last time, we also fit an additive linear model, which we found to work better than the tree. The test RMSE is lower, and the predicted vs actual plot looks much better.

```
bostom_lm = lm(medv ~ ., data = boston_trn)
boston_lm_tst_pred = predict(bostom_lm, newdata = boston_tst)
plot(boston_lm_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
```

```
     main = "Predicted vs Actual: Linear Model, Test Data"
)
abline(0, 1, col = "red", lwd = 2)
```




```
(lm_tst_rmse = rmse(boston_lm_tst_pred, boston_tst$medv))
```

```
## [1] 5.125877
```

27.1.3 Bagging

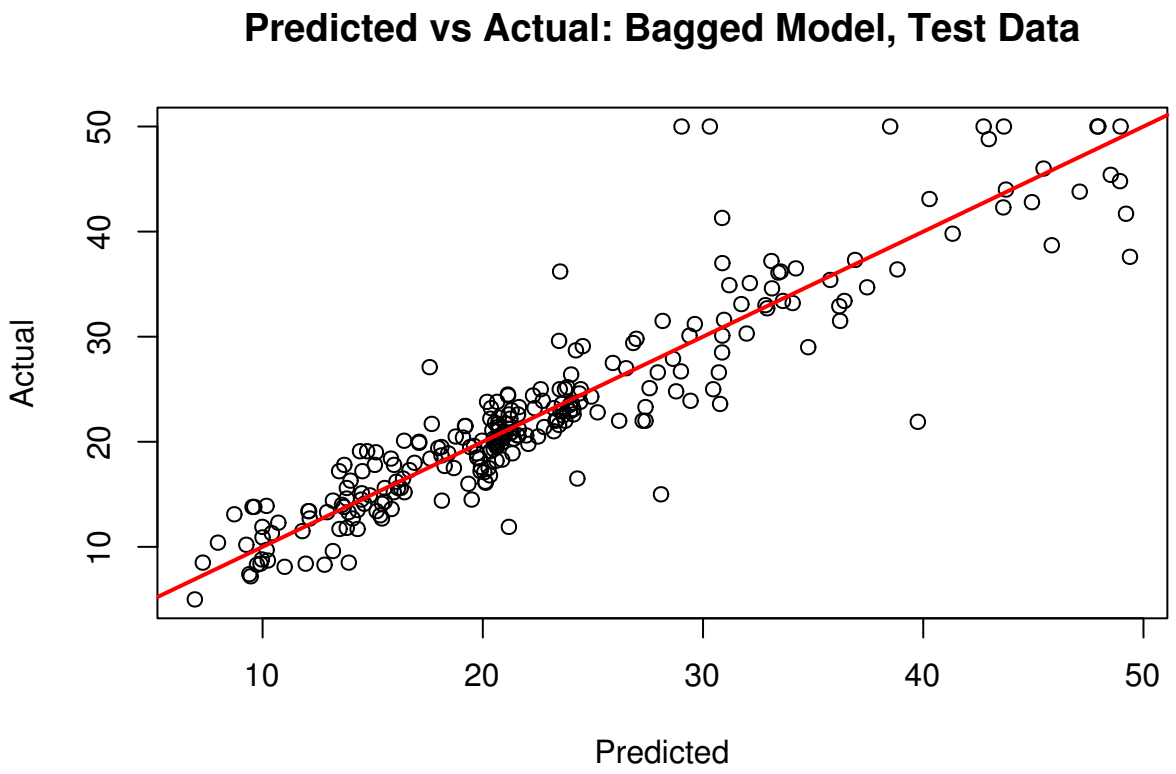
We now fit a bagged model, using the `randomForest` package. Bagging is actually a special case of a random forest where `mtry` is equal to p, the number of predictors.

```
boston_bag = randomForest(medv ~ ., data = boston_trn, mtry = 13,
                          importance = TRUE, ntrees = 500)
boston_bag
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston_trn, mtry = 13,      importance = TRUE, ntrees = 500)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
```

```
##
##           Mean of squared residuals: 14.20299
##                     % Var explained: 80.92
```

```
boston_bag_tst_pred = predict(boston_bag, newdata = boston_tst)
plot(boston_bag_tst_pred,boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Bagged Model, Test Data"
)
abline(0, 1, col = "red", lwd = 2)
```

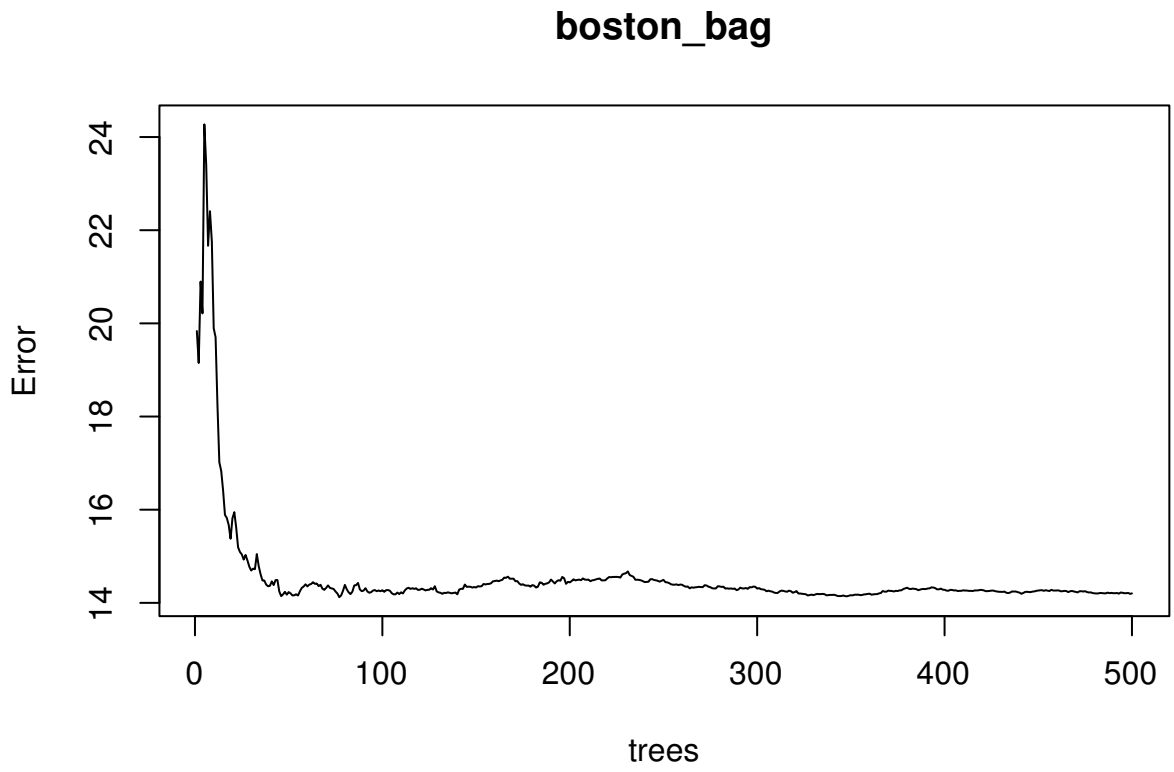



```
(bag_tst_rmse = rmse(boston_bag_tst_pred, boston_tst$medv))
```

```
## [1] 3.814368
```

Here we see two interesting results. First, the predicted versus actual plot no longer has a small number of predicted values. Second, our test error has dropped dramatically. Also note that the "Mean of squared residuals" which is output by `randomForest` is the **Out of Bag** estimate of the error.

```
plot(boston_bag)
```



27.1.4 Random Forest

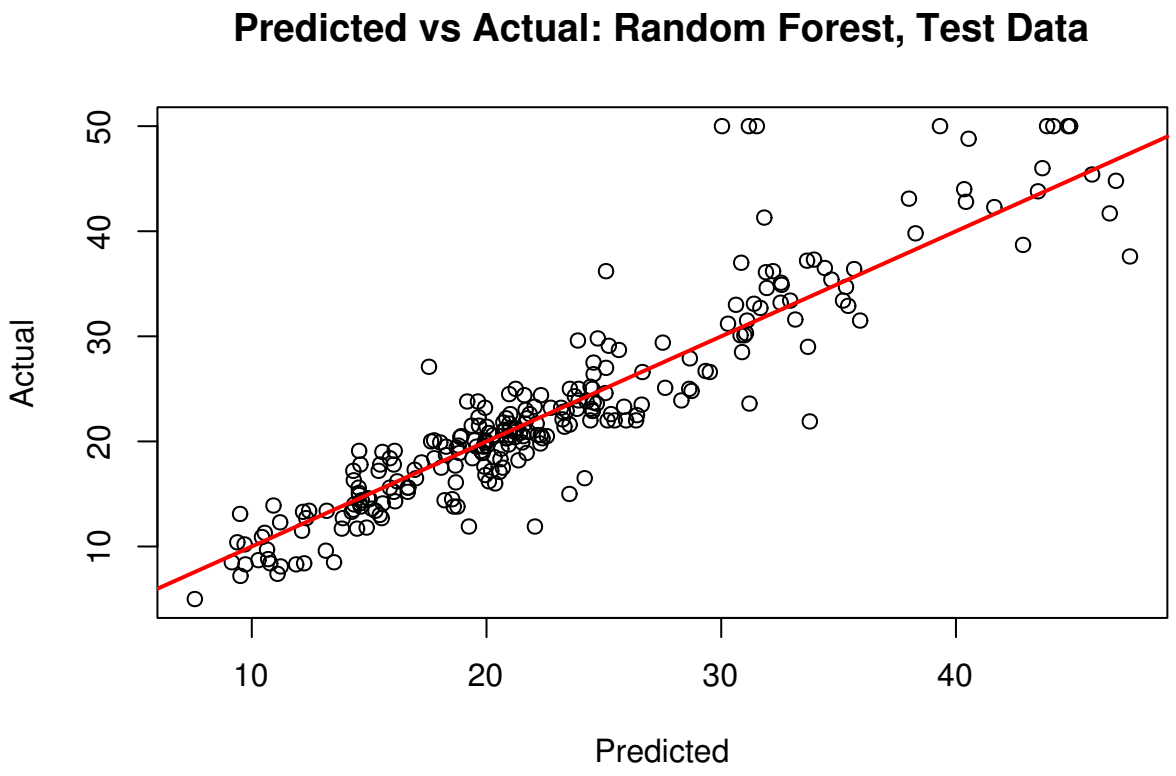
We now try a random forest. For regression, the suggestion is to use `mtry` equal to $p/3$.

```
boston_forest = randomForest(medv ~ ., data = boston_trn, mtry = 4,
                             importance = TRUE, ntrees = 500)
boston_forest
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston_trn, mtry = 4,      importance = TRUE, ntrees = 500)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 12.80737
##                     % Var explained: 82.79
```

```
#importance(boston_forest)
#varImpPlot(boston_forest)
boston_forest_tst_pred = predict(boston_forest, newdata = boston_tst)
plot(boston_forest_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Random Forest, Test Data"
```

```
)
abline(0, 1, col = "red", lwd = 2)
```




```
(forest_tst_rmse = rmse(boston_forest_tst_pred, boston_tst$medv))
```

```
## [1] 3.73447
```

```
boston_forest_trn_pred = predict(boston_forest, newdata = boston_trn)
forest_trn_rmse = rmse(boston_forest_trn_pred, boston_trn$medv)
forest_oob_rmse = rmse(boston_forest$predicted, boston_trn$medv)
```

Here we note three RMSEs. The training RMSE (which is optimistic), the OOB RMSE (which is a good estimate of the test error) and the test RMSE. Also note that variables importance was calculated, however, the results are not shown here. (The code to view the results is commented out.)

```
##       Data    Error
## 1 Training 1.563253
## 2      OOB 3.578738
## 3     Test 3.734470
```

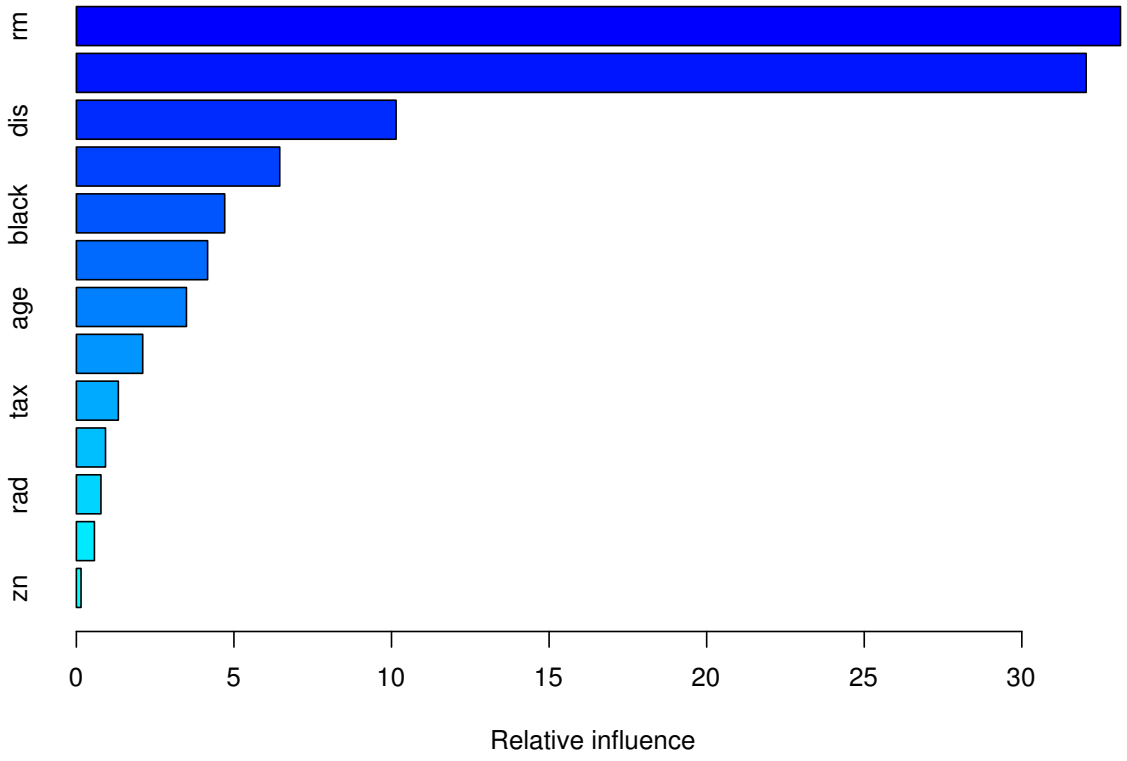
27.1.5 Boosting

Lastly, we try a boosted model, which by default will produce a nice **variable importance** plot as well as plots of the marginal effects of the predictors. We use the `gbm` package.

```
booston_boost = gbm(medv ~ ., data = boston_trn, distribution = "gaussian",
                    n.trees = 5000, interaction.depth = 4, shrinkage = 0.01)
booston_boost
```

```
## gbm(formula = medv ~ ., distribution = "gaussian", data = boston_trn,
##     n.trees = 5000, interaction.depth = 4, shrinkage = 0.01)
## A gradient boosted model with gaussian loss function.
## 5000 iterations were performed.
## There were 13 predictors of which 13 had non-zero influence.
```

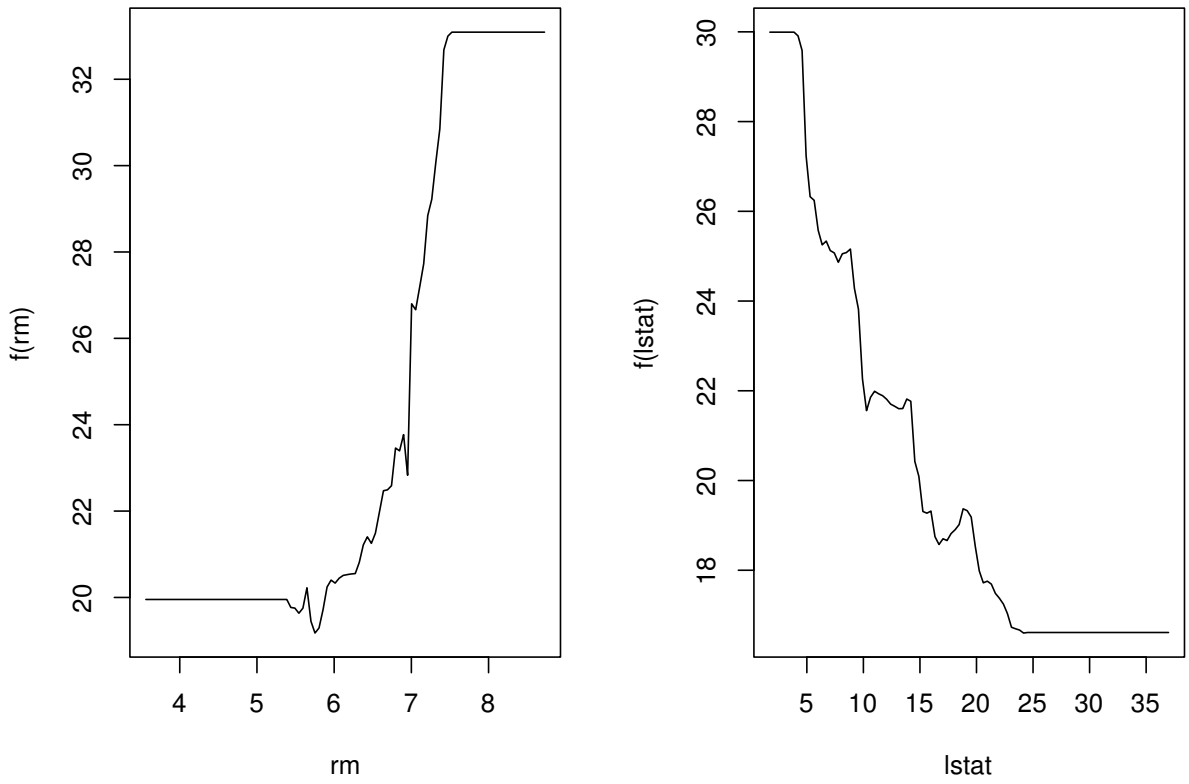
```
summary(booston_boost)
```




```
##             var    rel.inf
## rm           rm 33.1305117
## lstat     lstat 32.0413077
## dis         dis 10.1450348
## crim       crim  6.4535978
## black     black  4.7076459
## nox         nox  4.1647148
## age         age  3.4933055
## ptratio ptratio  2.1056374
## tax         tax  1.3320146
```

```
## indus     indus  0.9248479
## rad         rad  0.7800278
## chas       chas  0.5731764
## zn           zn  0.1481777
```

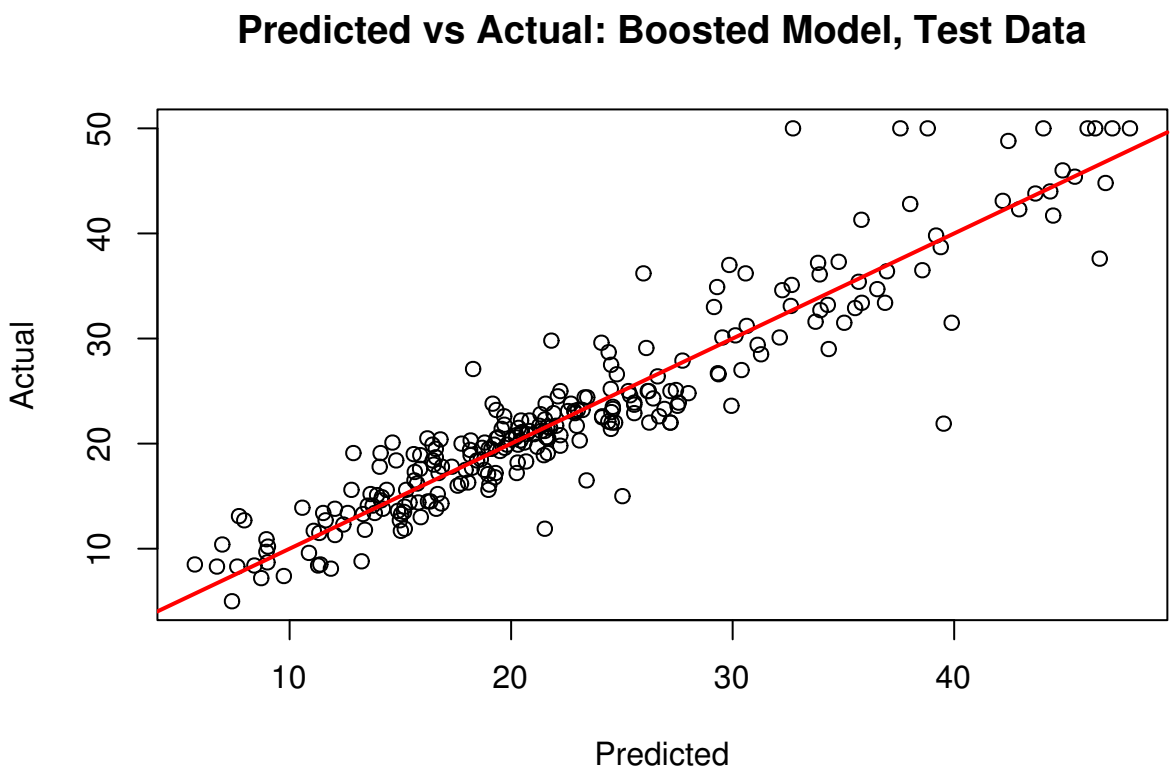
```
par(mfrow = c(1, 2))
plot(booston_boost, i = "rm")
plot(booston_boost, i = "lstat")
```




```
boston_boost_tst_pred = predict(booston_boost, newdata = boston_tst, n.trees = 5000)
(boost_tst_rmse = rmse(boston_boost_tst_pred, boston_tst$medv))
```

```
## [1] 3.437024
```

```
plot(boston_boost_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Boosted Model, Test Data"
)
abline(0, 1, col = "red", lwd = 2)
```



27.1.6 Results

```
(boston_rmse = data.frame(
  Model = c("Single Tree", "Linear Model", "Bagging",  "Random Forest",  "Boosting"),
  TestError = c(tree_tst_rmse, lm_tst_rmse, bag_tst_rmse, forest_tst_rmse, boost_tst_rmse)
  )
)
```

```
##           Model TestError
## 1   Single Tree  5.331457
## 2  Linear Model  5.125877
## 3       Bagging  3.814368
## 4 Random Forest  3.734470
## 5      Boosting  3.437024
```

While a single tree does not beat linear regression, each of the ensemble methods perform much better!

27.2 Classification

We now return to the `Carseats` dataset and the classification setting. We see that an additive logistic regression performs much better than a single tree, but we expect ensemble methods to bring trees closer to the logistic regression. Can they do better?

We now use prediction accuracy as our metric:

```
accuracy = function(actual, predicted) {
  mean(actual == predicted)
}
```

```
data(Carseats)
Carseats$Sales = as.factor(ifelse(Carseats$Sales <= 8, "Low", "High"))
set.seed(2)
seat_idx = sample(1:nrow(Carseats), 200)
seat_trn = Carseats[seat_idx,]
seat_tst = Carseats[-seat_idx,]
```

27.2.1 Tree Model

```
seat_tree = tree(Sales ~ ., data = seat_trn)
set.seed(3)
seat_tree_cv = cv.tree(seat_tree, FUN = prune.misclass)

seat_tree_prune = prune.misclass(seat_tree, best = 9)
seat_prune_tst_pred = predict(seat_tree_prune, seat_tst, type = "class")

table(predicted = seat_prune_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   60  22
##      Low    24  94
```

```
(tree_tst_acc = accuracy(predicted = seat_prune_tst_pred, actual = seat_tst$Sales))
```

```
## [1] 0.77
```

27.2.2 Logistic Regression

```
seat_glm = glm(Sales ~ ., data = seat_trn, family = "binomial")
seat_glm_tst_pred = ifelse(predict(seat_glm, seat_tst, "response") > 0.5,
                           "Low", "High")
table(predicted = seat_glm_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   75   9
##      Low     9 107
```

```
(glm_tst_acc = accuracy(predicted = seat_glm_tst_pred, actual = seat_tst$Sales))
```

```
## [1] 0.91
```

27.2.3 Bagging

```
seat_bag = randomForest(Sales ~ ., data = seat_trn, mtry = 10,
                        importance = TRUE, ntrees = 500)
seat_bag
```

```
##
## Call:
##  randomForest(formula = Sales ~ ., data = seat_trn, mtry = 10,      importance = TRUE, ntrees = 500)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 10
##
##         OOB estimate of  error rate: 21.5%
## Confusion matrix:
##      High Low class.error
## High   51  29   0.3625000
## Low    14 106   0.1166667
```

```
seat_bag_tst_pred = predict(seat_bag, newdata = seat_tst)
table(predicted = seat_bag_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   68  21
##      Low    16  95
```

```
(bag_tst_acc = accuracy(predicted = seat_bag_tst_pred, actual = seat_tst$Sales))
```

```
## [1] 0.815
```

27.2.4 Random Forest

For classification, the suggested `mtry` for a random forest is $\sqrt{p}$.

```
seat_forest = randomForest(Sales ~ ., data = seat_trn, mtry = 3, importance = TRUE, ntrees = 500)
seat_forest
```

```
##
## Call:
##  randomForest(formula = Sales ~ ., data = seat_trn, mtry = 3,      importance = TRUE, ntrees = 500)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 22%
## Confusion matrix:
##      High Low class.error
## High   49  31   0.3875000
## Low    13 107   0.1083333
```

```
seat_forest_tst_perd = predict(seat_forest, newdata = seat_tst)
table(predicted = seat_forest_tst_perd, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   63  16
##      Low    21 100
```

```
(forest_tst_acc = accuracy(predicted = seat_forest_tst_perd, actual = seat_tst$Sales))
```

```
## [1] 0.815
```

27.2.5 Boosting

To perform boosting, we modify the response to be `0` and `1` to work with `gbm`. Later we will use `caret` to fit `gbm` models, which will avoid this annoyance.

```
seat_trn_mod = seat_trn
seat_trn_mod$Sales = as.numeric(ifelse(seat_trn_mod$Sales == "Low", "0", "1"))

seat_boost = gbm(Sales ~ ., data = seat_trn_mod, distribution = "bernoulli",
                 n.trees = 5000, interaction.depth = 4, shrinkage = 0.01)
seat_boost
```

```
## gbm(formula = Sales ~ ., distribution = "bernoulli", data = seat_trn_mod,
##     n.trees = 5000, interaction.depth = 4, shrinkage = 0.01)
## A gradient boosted model with bernoulli loss function.
## 5000 iterations were performed.
## There were 10 predictors of which 10 had non-zero influence.
```

```
seat_boost_tst_pred = ifelse(predict(seat_boost, seat_tst, n.trees = 5000, "response") > 0.5,
                             "High", "Low")
table(predicted = seat_boost_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   70  17
##      Low    14  99
```

```
(boost_tst_acc = accuracy(predicted = seat_boost_tst_pred, actual = seat_tst$Sales))
```

```
## [1] 0.845
```

27.2.6 Results

```
(seat_acc = data.frame(
  Model = c("Single Tree", "Logistic Regression", "Bagging",  "Random Forest",  "Boosting"),
  TestAccuracy = c(tree_tst_acc, glm_tst_acc, bag_tst_acc, forest_tst_acc, boost_tst_acc)
  )
)
```

```
##                 Model TestAccuracy
## 1         Single Tree        0.770
## 2 Logistic Regression        0.910
## 3             Bagging        0.815
## 4       Random Forest        0.815
## 5            Boosting        0.845
```

Here we see each of the ensemble methods performing better than a single tree, however, they still fall behind logistic regression. Sometimes a simple linear model will beat more complicated models! This is why you should always try a logistic regression for classification.

27.3 Tuning

So far we fit bagging, boosting and random forest models, but did not tune any of them, we simply used certain, somewhat arbitrary, parameters. Now we will see how to modify the tuning parameters to make these models better.

- Bagging: Actually just a subset of Random Forest with `mtry` $= p$.
- Random Forest: `mtry`
- Boosting: `n.trees`, `interaction.depth`, `shrinkage`, `n.minobsinnode`

We will use the `caret` package to accomplish this. Technically `ntrees` is a tuning parameter for both bagging and random forest, but `caret` will use 500 by default and there is no easy way to tune it. This will not make a big difference since for both we simply need "enough" and 500 seems to do the trick.

While `mtry` is a tuning parameter, there are suggested values for classification and regression:

- Regression: `mtry` $= p/3$.
- Classification: `mtry` $= \sqrt{p}$.

Also note that with these tree-based ensemble methods there are two resampling solutions for tuning the model:

- Out of Bag
- Cross-Validation

Using Out of Bag samples is advantageous with these methods as compared to Cross-Validation since it removes the need to refit the model and is thus much more computationally efficient. Unfortunately OOB methods cannot be used with `gbm` models. See the `caret` documentation for details.

27.3.1 Random Forest and Bagging

Here we setup training control for both OOB and cross-validation methods. Note we specify `verbose = FALSE` which suppresses output related to progress. You may wish to set this to `TRUE` when first tuning a model since it will give you an idea of how long the tuning process will take. (Which can sometimes be a long time.)

```
oob = trainControl(method = "oob")
cv_5 = trainControl(method = "cv", number = 5)
```

To tune a Random Forest in `caret` we will use `method = "rf"` which uses the `randomForest` function in the background. Here we elect to use the OOB training control that we created. We could also use Cross-Validation, however it will likely select a similar model, but requiring more time.

We setup a grid of `mtry` values which include all possible values since there are 10 predictors in the dataset. An `mtry` of 10 is actually bagging.

```
dim(seat_trn)
```

```
## [1] 200  11
```

```
rf_grid =  expand.grid(mtry = 1:10)
set.seed(825)
seat_rf_tune = train(Sales ~ ., data = seat_trn,
                     method = "rf",
                     trControl = oob,
                     verbose = FALSE,
                     tuneGrid = rf_grid)
seat_rf_tune
```

```
## Random Forest
##
## 200 samples
##  10 predictor
##   2 classes: 'High', 'Low'
##
## No pre-processing
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##    1    0.730     0.3807339
##    2    0.775     0.5033113
##    3    0.795     0.5591398
##    4    0.790     0.5474138
##    5    0.795     0.5610278
##    6    0.790     0.5512821
##    7    0.805     0.5806452
##    8    0.795     0.5628998
##    9    0.785     0.5376344
##   10    0.775     0.5202559
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 7.
```

```
accuracy(predict(seat_rf_tune, seat_tst), seat_tst$Sales)
```

```
## [1] 0.795
```

The results returned are based on the OOB samples. (Coincidentally, the test accuracy is the same as the best accuracy found using OOB samples.) Note that when using OOB, for some reason the default plot is not what you would expect and is not at all useful. (Which is why it is omitted here.)

```
seat_rf_tune$bestTune
```

```
##   mtry
## 7    7
```

Based on these results, we would select the random forest model with an `mtry` of 7. Note that based on the OOB estimates, the bagging model is expected to perform worse than this select model, however, based on our results above, that is not what we find to be true in our test set.

Also note that `method = "ranger"` would also fit a random forest model. Ranger is a newer `R` package for random forests that has been shown to be much faster, especially when there are a larger number of predictors.

27.3.2 Boosting

We now tune a boosted tree model. We will use the cross-validation tune control setup above. We will fit the model using `gbm` with `caret`.

To setup the tuning grid, we must specify four parameters to tune:

- `interaction.depth`: How many splits to use with each tree.
- `n.trees`: The number of trees to use.
- `shrinkage`: The shrinkage parameters, which controls how fast the method learns.
- `n.minobsinnode`: The minimum number of observations in a node of the tree. (`caret` requires us to specify this. This is actually a tuning parameter of the trees, not boosting, and we would normally just accept the default.)

Finally, `expand.grid` comes in handy, as we can specify a vector of values for each parameter, then we get back a matrix of all possible combinations.

```
gbm_grid =  expand.grid(interaction.depth = 1:5,
                        n.trees = (1:6) * 500,
                        shrinkage = c(0.001, 0.01, 0.1),
                        n.minobsinnode = 10)
```

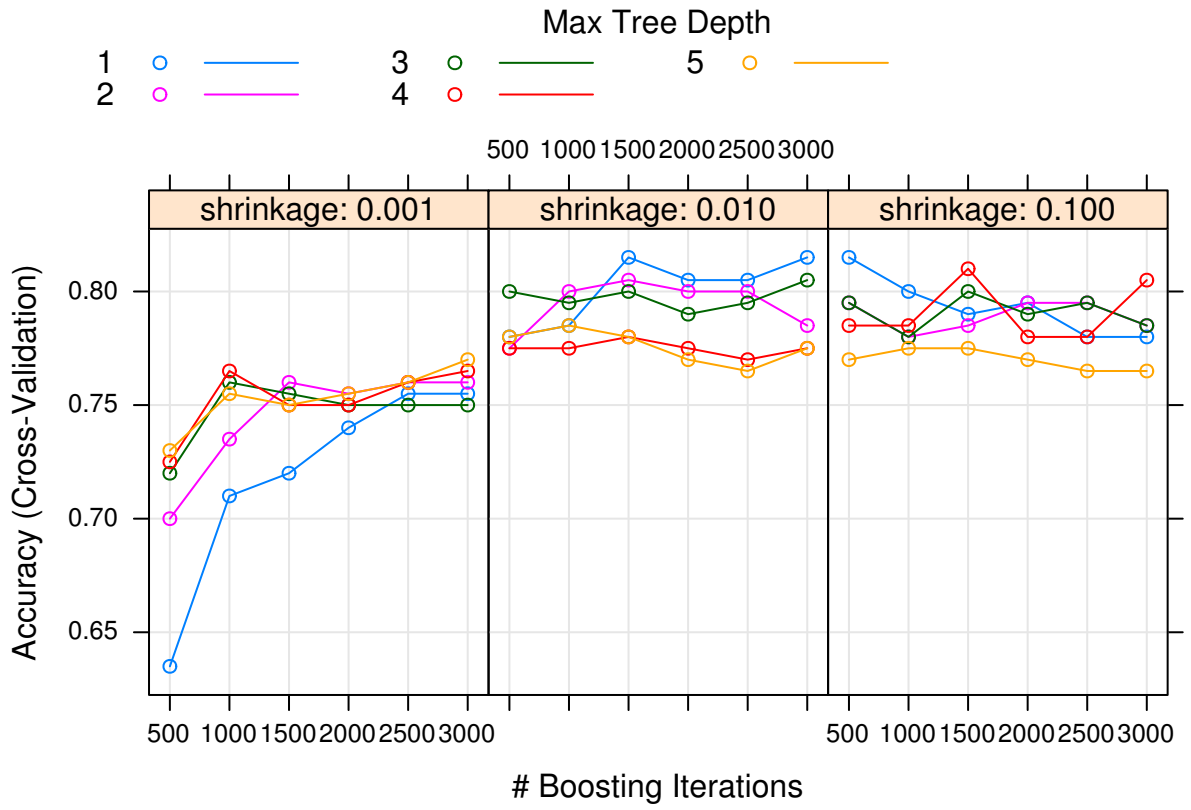
We now train the model using all possible combinations of the tuning parameters we just specified.

```
seat_gbm_tune = train(Sales ~ ., data = seat_trn,
                      method = "gbm",
                      trControl = cv_5,
                      verbose = FALSE,
                      tuneGrid = gbm_grid)
```

The additional `verbose = FALSE` in the `train` call suppresses additional output from each `gbm` call.

By default, calling `plot` here will produce a nice graphic of the results.

```
#seat_gbm_tune
plot(seat_gbm_tune)
```




```
accuracy(predict(seat_gbm_tune, seat_tst), seat_tst$Sales)
```

```
## [1] 0.845
```

We see our tuned model does no better on the test set than the arbitrary boosted model we had fit above, with the slightly different parameters seen below. We could perhaps try a larger tuning grid, but at this point it seems unlikely that we could find a much better model. There seems to be no way to get a tree method to out-perform logistic regression in this dataset.

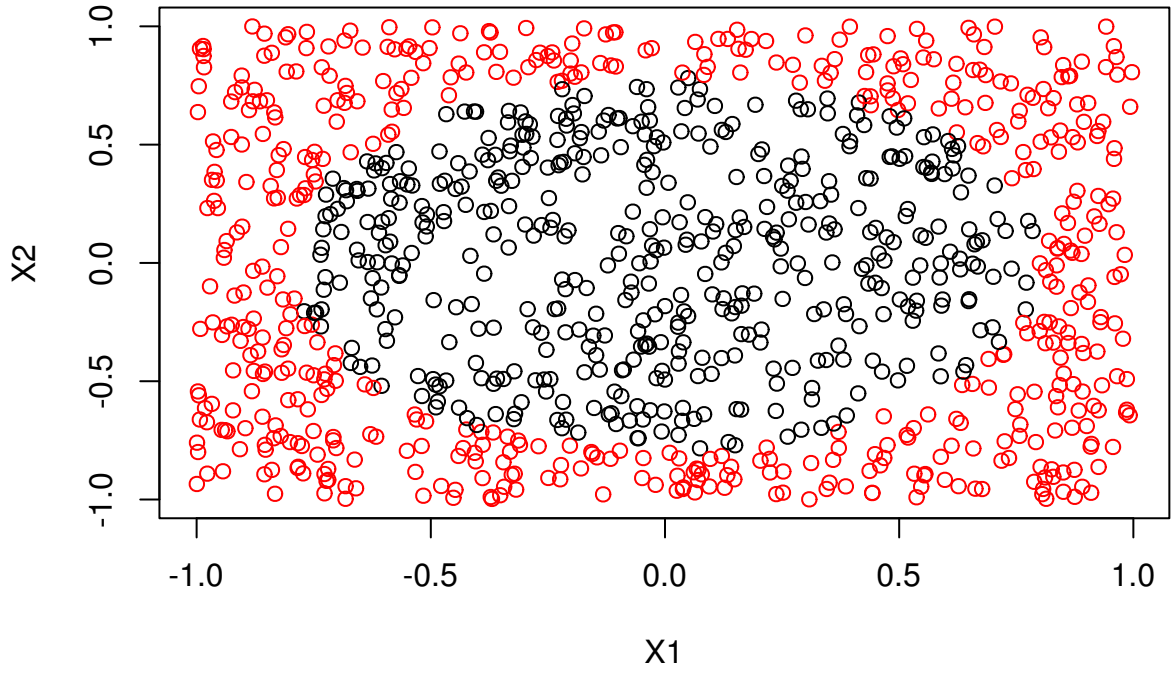
```
seat_gbm_tune$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 61     500                 1       0.1             10
```

27.4 Tree versus Ensemble Boundaries

```
library(mlbench)
set.seed(42)
sim_trn = mlbench.circle(n = 1000, d = 2)
sim_trn = data.frame(sim_trn$x, class = as.factor(sim_trn$classes))
sim_tst = mlbench.circle(n = 1000, d = 2)
sim_tst = data.frame(sim_tst$x, class = as.factor(sim_tst$classes))
```

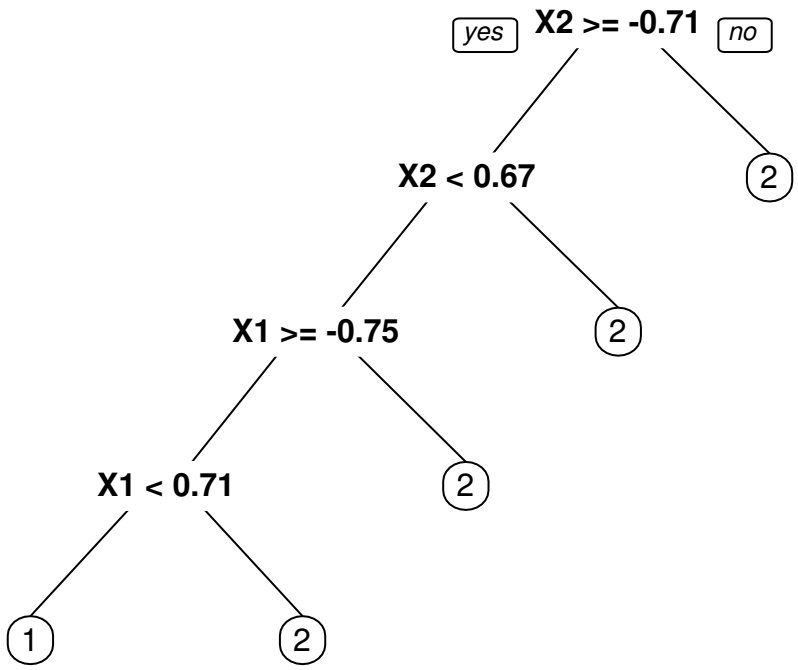
```
plot(sim_trn$X1, sim_trn$X2, col = sim_trn$class,
     xlab = "X1", ylab = "X2")
```




```
cv_5 = trainControl(method = "cv", number = 5)
oob  = trainControl(method = "oob")
```

```
sim_tree_cv = train(class ~ .,
                    data = sim_trn,
                    trControl = cv_5,
                    method = "rpart")
```

```
library(rpart.plot)
prp(sim_tree_cv$finalModel)
```



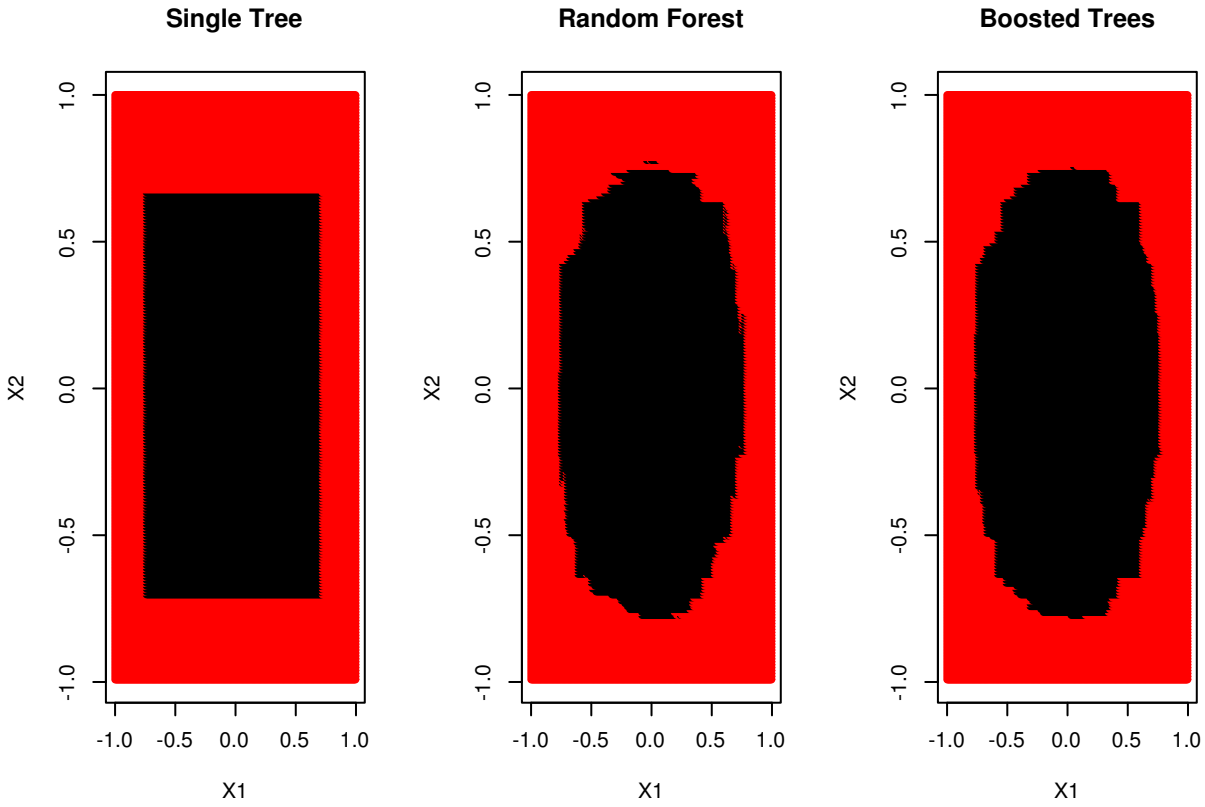

```
rf_grid = expand.grid(mtry = c(1, 2))
sim_rf_oob = train(class ~ .,
                   data = sim_trn,
                   trControl = oob,
                   tuneGrid = rf_grid)
```

```
gbm_grid =  expand.grid(interaction.depth = 1:5,
                        n.trees = (1:6) * 500,
                        shrinkage = c(0.001, 0.01, 0.1),
                        n.minobsinnode = 10)

sim_gbm_cv = train(class ~ .,
                   data = sim_trn,
                   method = "gbm",
                   trControl = cv_5,
                   verbose = FALSE,
                   tuneGrid = gbm_grid)
```

```
plot_grid = expand.grid(
  X1 = seq(min(sim_tst$X1), max(sim_tst$X1), by = 0.01),
  X2 = seq(min(sim_tst$X2), max(sim_tst$X2), by = 0.01)
)

tree_pred = predict(sim_tree_cv, plot_grid)
rf_pred   = predict(sim_rf_oob, plot_grid)
```

```
gbm_pred  = predict(sim_gbm_cv, plot_grid)
```

```
par(mfrow = c(1, 3))
plot(plot_grid$X1, plot_grid$X2, col = tree_pred,
     xlab = "X1", ylab = "X2", pch = 20, main = "Single Tree")
plot(plot_grid$X1, plot_grid$X2, col = rf_pred,
     xlab = "X1", ylab = "X2", pch = 20, main = "Random Forest")
plot(plot_grid$X1, plot_grid$X2, col = gbm_pred,
     xlab = "X1", ylab = "X2", pch = 20, main = "Boosted Trees")
```



27.5 External Links

- Classification and Regression by `randomForest` - Introduction to the `randomForest` package in `R` news.
- `ranger`: A Fast Implementation of Random Forests - Alternative package for fitting random forests with potentially better speed.
- On `ranger`'s respect.unordered.factors Argument - A note on handling of categorical variables with random forests.
- Extremely Randomized Trees
- `extraTrees` Method for Classificationand Regression
- XGBoost - Scalable and Flexible Gradient Boosting
- XGBoost `R` Tutorial

27.6 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.4.1 and the following packages:

- Base Packages, Attached

```
## [1] "methods"   "parallel"  "splines"   "stats"     "graphics"  "grDevices"
## [7] "utils"     "datasets"  "base"
```

- Additional Packages, Attached

```
##  [1] "rpart.plot"   "rpart"        "mlbench"      "plyr"
##  [5] "caret"        "ggplot2"      "gbm"          "lattice"
##  [9] "survival"     "randomForest" "ISLR"         "MASS"
## [13] "tree"
```

- Additional Packages, Not Attached

```
##  [1] "Rcpp"         "lubridate"    "class"        "assertthat"
##  [5] "rprojroot"    "digest"       "ipred"        "foreach"
##  [9] "R6"           "backports"    "stats4"       "evaluate"
## [13] "e1071"        "rlang"        "lazyeval"     "kernlab"
## [17] "Matrix"       "rmarkdown"    "CVST"         "ddalpha"
## [21] "gower"        "stringr"      "munsell"      "compiler"
## [25] "pkgconfig"    "dimRed"       "htmltools"    "nnet"
## [29] "tibble"       "prodlim"      "DRR"          "bookdown"
## [33] "codetools"    "RcppRoll"     "dplyr"        "withr"
## [37] "recipes"      "ModelMetrics" "grid"         "nlme"
## [41] "gtable"       "magrittr"     "scales"       "stringi"
## [45] "reshape2"     "bindrcpp"     "timeDate"     "robustbase"
## [49] "lava"         "iterators"    "tools"        "glue"
## [53] "DEoptimR"     "purrr"        "yaml"         "colorspace"
## [57] "knitr"        "bindr"
```

Chapter 28

Artificial Neural Networks

Part VII

Appendix

Chapter 29

Overview

TODO: Add a section about "coding" tips and tricks. For example: beware when using code you found on the internet.

TODO: Add a section about ethics in machine learning

- https://www.newyorker.com/news/daily-comment/the-ai-gaydar-study-and-the-real-dangers-of-big-data
- https://www.propublica.org/article/facebook-enabled-advertisers-to-reach-jew-haters

Chapter 30

Non-Linear Models

TOOD: This chpater is currently empty to reduce build time.

Chapter 31

Regularized Discriminant Analysis

TOOD: This chpater is currently empty to reduce build time.

Chapter 32

Support Vector Machines

TOOD: This chpater is currently empty to reduce build time.

Bibliography