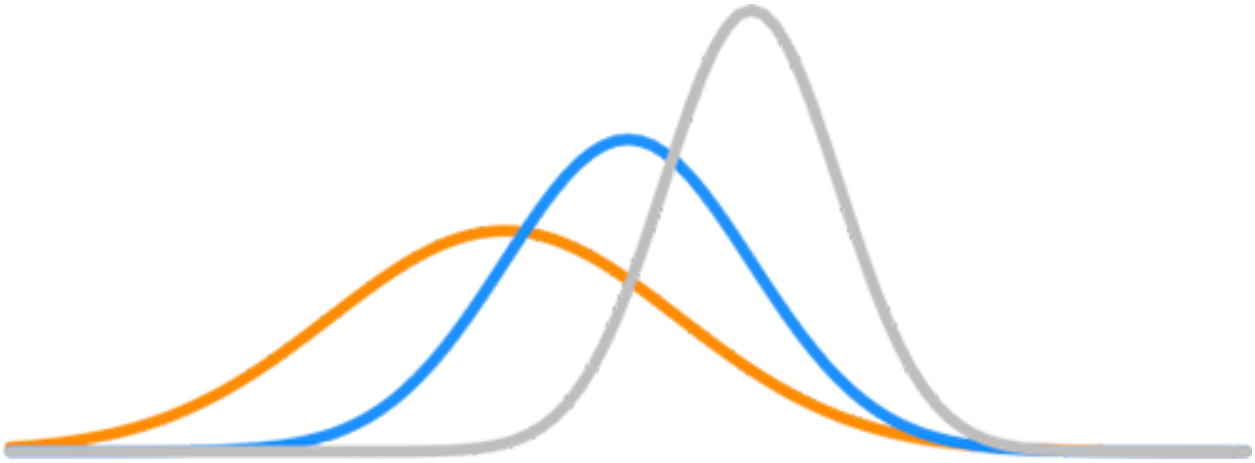
R for Statistical Learning

David Dalpiaz

2018-11-20

Contents

Introduction



Welcome to R for Statistical Learning! While this is the current title, a more appropriate title would be "Machine Learning from the Perspective of a Statistician using R" but that doesn't seem as catchy.

About This Book

This book currently serves as a supplement to An Introduction to Statistical Learning for STAT 432 - Basics of Statistical Learning at the University of Illinois at Urbana-Champaign.

The initial focus of this text was to expand on ISL's introduction to using R for statistical learning, mostly through adding to and modifying existing code. This text is currently becoming much more self-contained. Why? A very good question considering that the author consider ISL one of the best undergraduate textbooks available, and was one of the driving forces for the creation of STAT 432. However once the course was created, exact control over content became extremely useful. The main focus of this text is to match the needs of students in that course. Some of those needs include:

- Additional R code examples and explanation
- Simulation studies
- Mathematical rigor that matches the background of the readers
- A book structure that matches the overall structure of the course

In other words, this text seeks to replicate the best parts of An Introduction to Statistical Learning, The Elements of Statistical Learning, and Applied Predictive Modeling that are most needed by a particular set of students.

Organization

The text is organized into roughly 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. It is currently rather sparse.

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.

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.

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 are occurring 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. While development is taking place, formatting in the pdf version may not be as well planned as the html version since the html version does not need to worry about pagination.

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.

While development is taking place, you may 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.

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.

Acknowledgements

The following is a (likely incomplete) list of helpful contributers.

- James Balamuta, Summer 2016 - ???
- Korawat Tanwisuth, Spring 2017
- Yiming Gao, Spring 2017
- Binxiang Ni, Summer 2017
- Ruiqi (Zoe) Li, Summer 2017
- Haitao Du, Summer 2017
- Rachel Banoff, Fall 2017
- Chenxing Wu, Fall 2017
- Wenting Xu, Fall 2017
- Yuanning Wei, Fall 2017
- Ross Drucker, Fall 2017
- Craig Bonsignore, Fall 2018

Your name could be here! 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. Pull requests encouraged!

Looking for ways to contribute?

- You'll notice that a lot of the plotting code is not displayed in the text, but is availible in the source. Currently that code was written to accomplish a task, but without much thought about the best way to accomplish the task. Try refactoring some of this code.
- Fix typos. Since the book is actively being developed, typos are getting added all the time.
- Suggest edits. Good feedback can be just as helpful as actually contributing code changes.

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

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** of 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 learning 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.   37.8      69.2  22.1
##  2  44.5  39.3      45.1  10.4
##  3  17.2  45.9      69.3   9.3
##  4 152.   41.3      58.5  18.5
##  5 181.   10.8      58.4  12.9
##  6   8.7  48.9      75     7.2
##  7  57.5  32.8      23.5  11.8
##  8 120.   19.6      11.6  13.2
##  9   8.6   2.1       1     4.8
## 10 200.    2.6      21.2  10.6
## # ... with 190 more rows
```

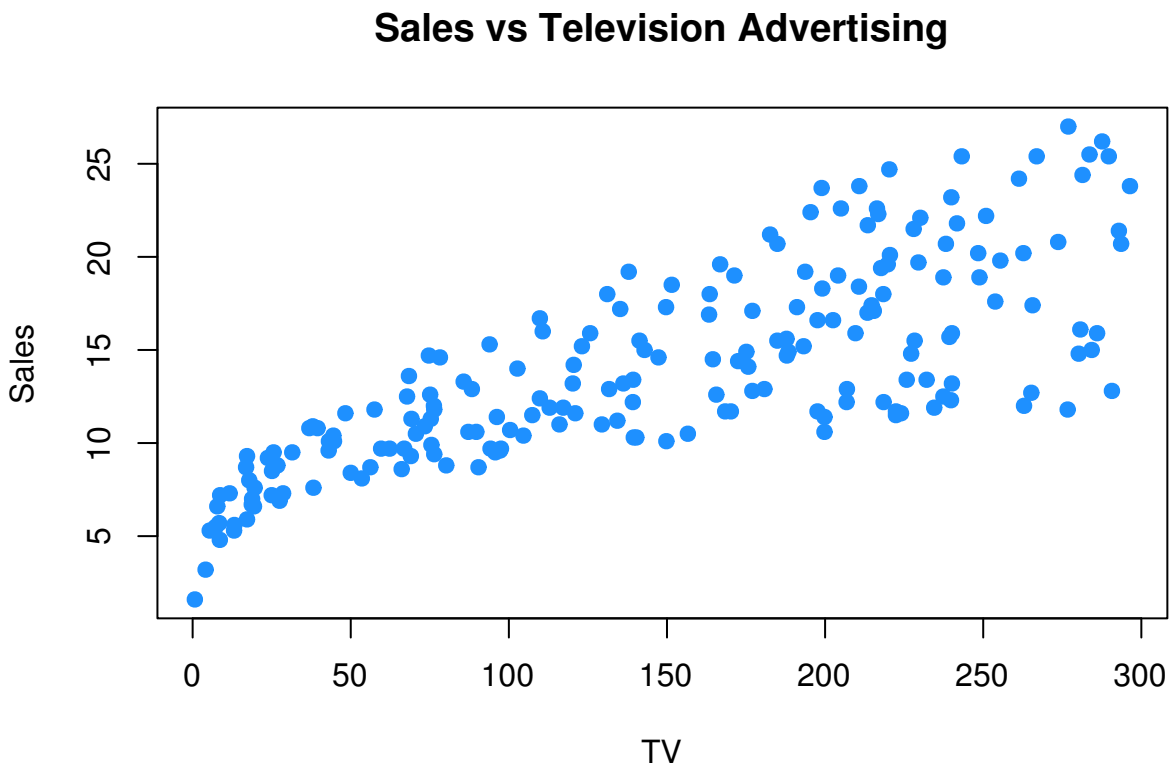
Because the data was read using `read_csv()`, `Advertising` is a tibble. We see that there are a total of 200 observations and 4 variables, each of which is numeric. (Specifically double-precision vectors, but more importantly they are numbers.) For the purpose of this analysis, `Sales` will be the **response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

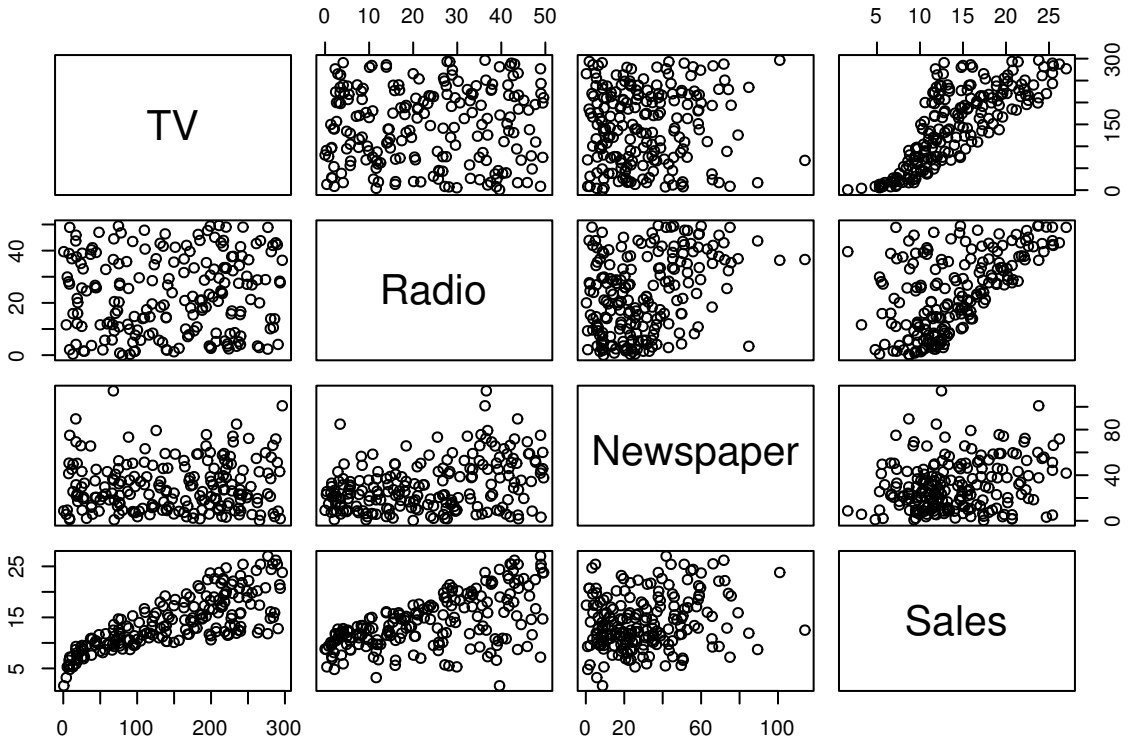
We could do so for any individual predictor.

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



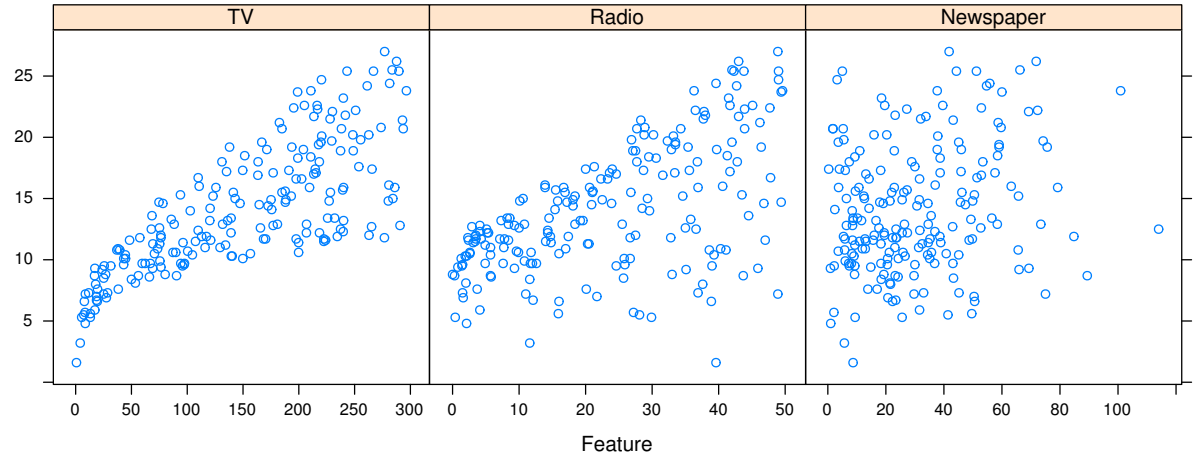
The `pairs()` function is a useful way to quickly visualize a number of scatter plots.

```
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
##    <fct> <dbl>   <dbl>
##  1 A     0.129 -0.377
##  2 A     0.225  0.145
##  3 A     0.437  0.799
##  4 A     0.654  1.70
##  5 A     0.345 -0.747
##  6 A     0.368 -0.196
##  7 A     0.197  1.43
##  8 A     0.690 -0.0139
##  9 A     0.127 -2.11
## 10 A     0.927 -1.16
## # ... 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
##  -0.2035111   0.4110860  -0.1340925   0.3674488
```

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

```
## (Intercept)         x1B         x1C          x2      x1B:x2      x1C:x2
##  -0.2161457   0.7738502  -0.5310440   0.3982779  -0.7671523   0.6726809
```

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 3.5.1. The following packages (and their dependencies) were loaded in this file:

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

Part II

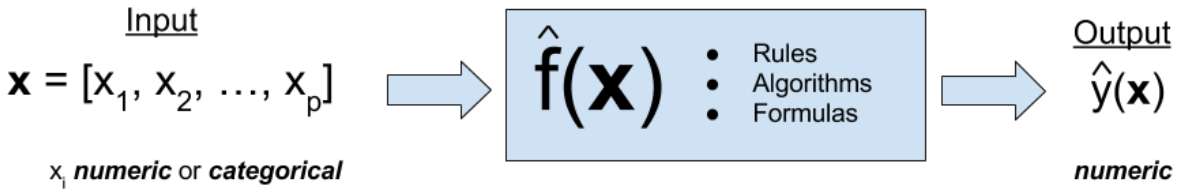
Regression

Chapter 5

Overview

Chapter Status: This chapter is currently undergoing massive rewrites. Some code is mostly for the use of current students. Plotting code is often not best practice for plotting, but is instead useful for understanding.

- **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.



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.

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

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

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

- TODO: RSS vs R^2 vs RMSE

Code for Plotting from Class

```
## load packages
library(rpart)
library(FNN)
```

```
# simulate data

## signal
f = function(x) {
  x ^ 3
}

## define data generating processs
get_sim_data = function(f, sample_size = 50) {
  x = runif(n = sample_size, min = -1, max = 1)
  y = rnorm(n = sample_size, mean = f(x), sd = 0.15)
  data.frame(x, y)
}

## simualte training data
set.seed(42)
sim_trn_data = get_sim_data(f = f)

## simulate testing data
set.seed(3)
sim_tst_data = get_sim_data(f = f)
```

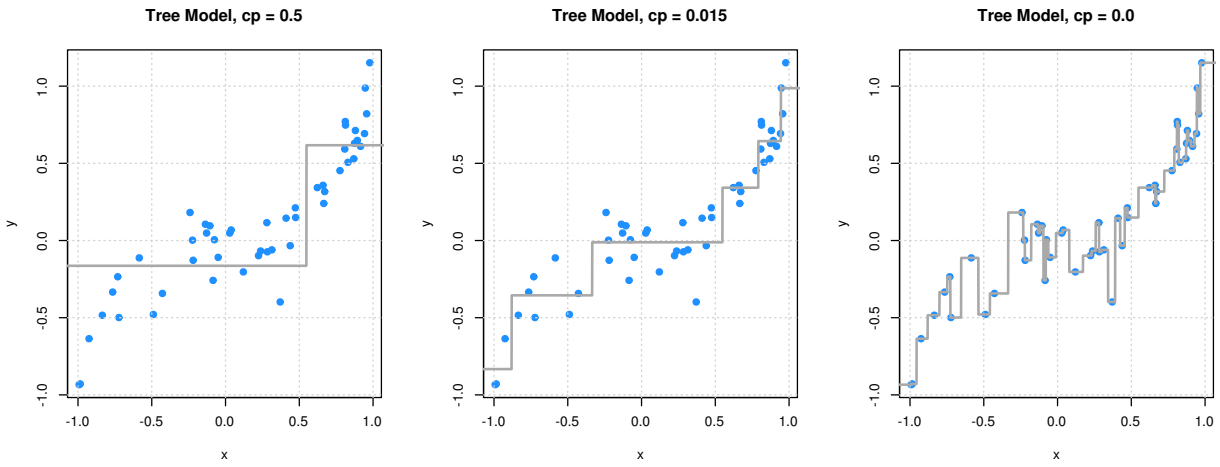
```
## create grid for plotting
x_grid = data.frame(x = seq(-1.5, 1.5, 0.001))
```

```
# fit models

## tree models
tree_fit_l = rpart(y ~ x, data = sim_trn_data,
                   control = rpart.control(cp = 0.500, minsplit = 2))
tree_fit_m = rpart(y ~ x, data = sim_trn_data,
                   control = rpart.control(cp = 0.015, minsplit = 2))
tree_fit_h = rpart(y ~ x, data = sim_trn_data,
                   control = rpart.control(cp = 0.000, minsplit = 2))

## knn models
knn_fit_l = knn.reg(train = sim_trn_data["x"], y = sim_trn_data$y,
                    test = x_grid, k = 40)
knn_fit_m = knn.reg(train = sim_trn_data["x"], y = sim_trn_data$y,
                    test = x_grid, k = 5)
knn_fit_h = knn.reg(train = sim_trn_data["x"], y = sim_trn_data$y,
                    test = x_grid, k = 1)

## polynomial models
poly_fit_l = lm(y ~ poly(x,  1), data = sim_trn_data)
poly_fit_m = lm(y ~ poly(x,  3), data = sim_trn_data)
poly_fit_h = lm(y ~ poly(x, 22), data = sim_trn_data)
```

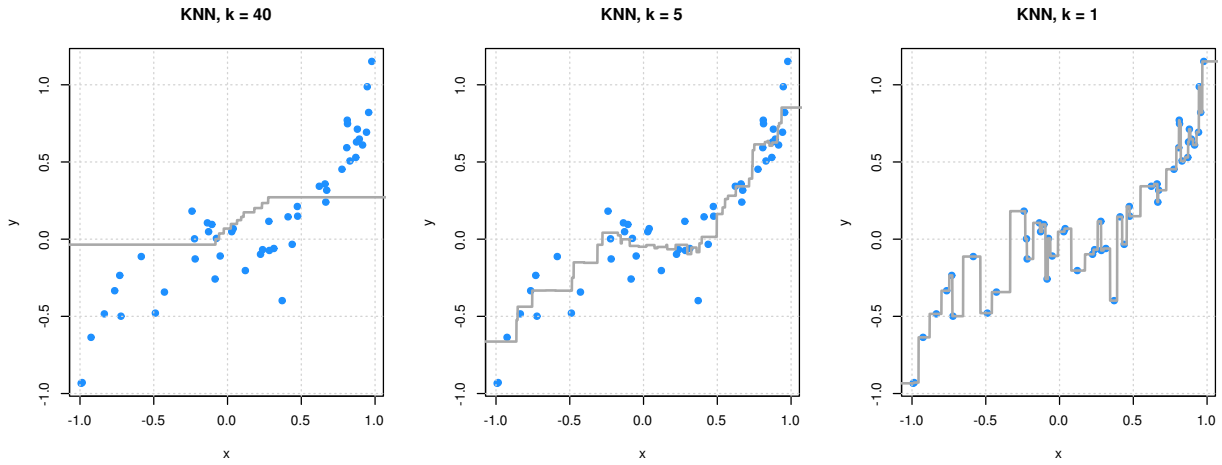
```
# get predictions

## tree models
tree_fit_l_pred = predict(tree_fit_l, newdata = x_grid)
tree_fit_m_pred = predict(tree_fit_m, newdata = x_grid)
tree_fit_h_pred = predict(tree_fit_h, newdata = x_grid)

## knn models
knn_fit_l_pred = knn_fit_l$pred
knn_fit_m_pred = knn_fit_m$pred
knn_fit_h_pred = knn_fit_h$pred

## polynomial models
poly_fit_l_pred = predict(poly_fit_l, newdata = x_grid)
poly_fit_m_pred = predict(poly_fit_m, newdata = x_grid)
poly_fit_h_pred = predict(poly_fit_h, newdata = x_grid)
```

```
# plot fitted trees

par(mfrow = c(1, 3))
plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Tree Model, cp = 0.5", cex = 1.5)
grid()
lines(x_grid$x, tree_fit_l_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Tree Model, cp = 0.015", cex = 1.5)
grid()
```

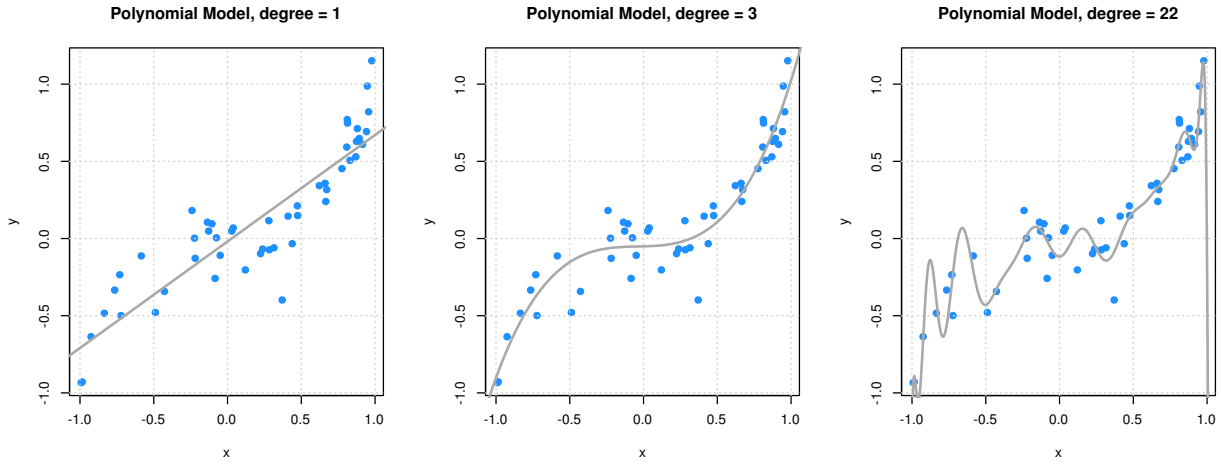
```
lines(x_grid$x, tree_fit_m_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Tree Model, cp = 0.0", cex = 1.5)
grid()
lines(x_grid$x, tree_fit_h_pred, col = "darkgrey",  lwd = 2)
```




```
# plot fitted KNN

par(mfrow = c(1, 3))
plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "KNN, k = 40", cex = 1.5)
grid()
lines(x_grid$x, knn_fit_l_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "KNN, k = 5", cex = 1.5)
grid()
lines(x_grid$x, knn_fit_m_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "KNN, k = 1", cex = 1.5)
grid()
lines(x_grid$x, knn_fit_h_pred, col = "darkgrey",  lwd = 2)
```




```
# plot fitted polynomials
par(mfrow = c(1, 3))
plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Polynomial Model, degree = 1", cex = 1.5)
grid()
lines(x_grid$x, poly_fit_l_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Polynomial Model, degree = 3", cex = 1.5)
grid()
lines(x_grid$x, poly_fit_m_pred, col = "darkgrey",  lwd = 2)

plot(y ~ x, data = sim_trn_data, col = "dodgerblue", pch = 20,
     main = "Polynomial Model, degree = 22", cex = 1.5)
grid()
lines(x_grid$x, poly_fit_h_pred, col = "darkgrey",  lwd = 2)
```



Chapter 6

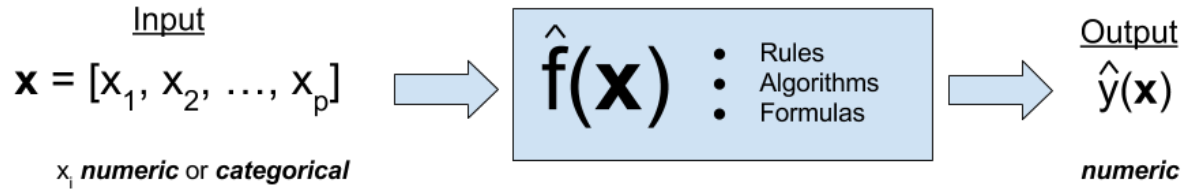
Linear Models

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.



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

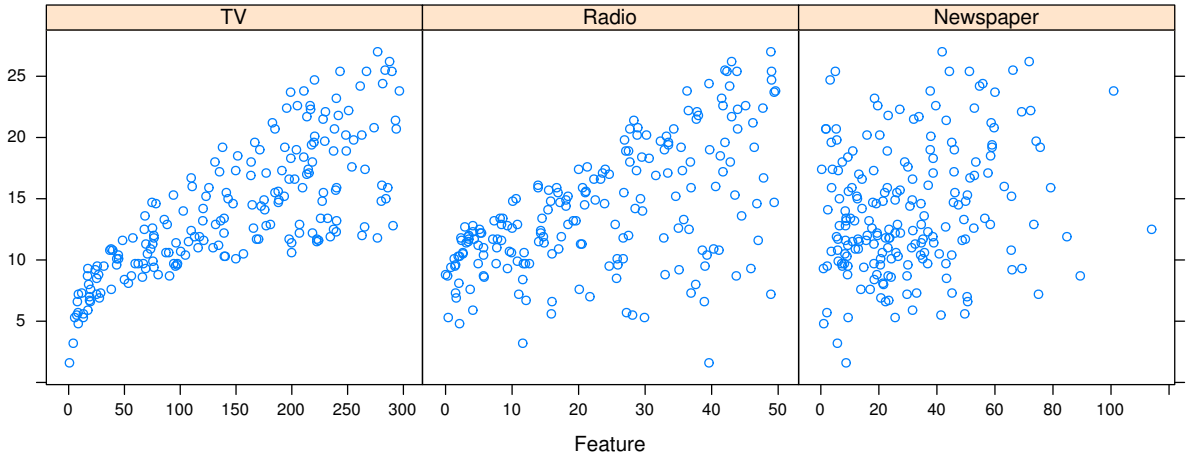
To discuss linear models in the context of prediction, we return to the `Advertising` data from the previous chapter.

```
Advertising
```

```
## # A tibble: 200 x 4
##       TV Radio Newspaper Sales
##    <dbl> <dbl>     <dbl> <dbl>
##  1 230.   37.8      69.2  22.1
##  2  44.5  39.3      45.1  10.4
##  3  17.2  45.9      69.3   9.3
##  4 152.   41.3      58.5  18.5
##  5 181.   10.8      58.4  12.9
##  6   8.7  48.9      75     7.2
##  7  57.5  32.8      23.5  11.8
##  8 120.   19.6      11.6  13.2
##  9   8.6   2.1       1     4.8
## 10 200.    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.

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

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 consider 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 = subset(data, select = response, drop = TRUE),
       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.

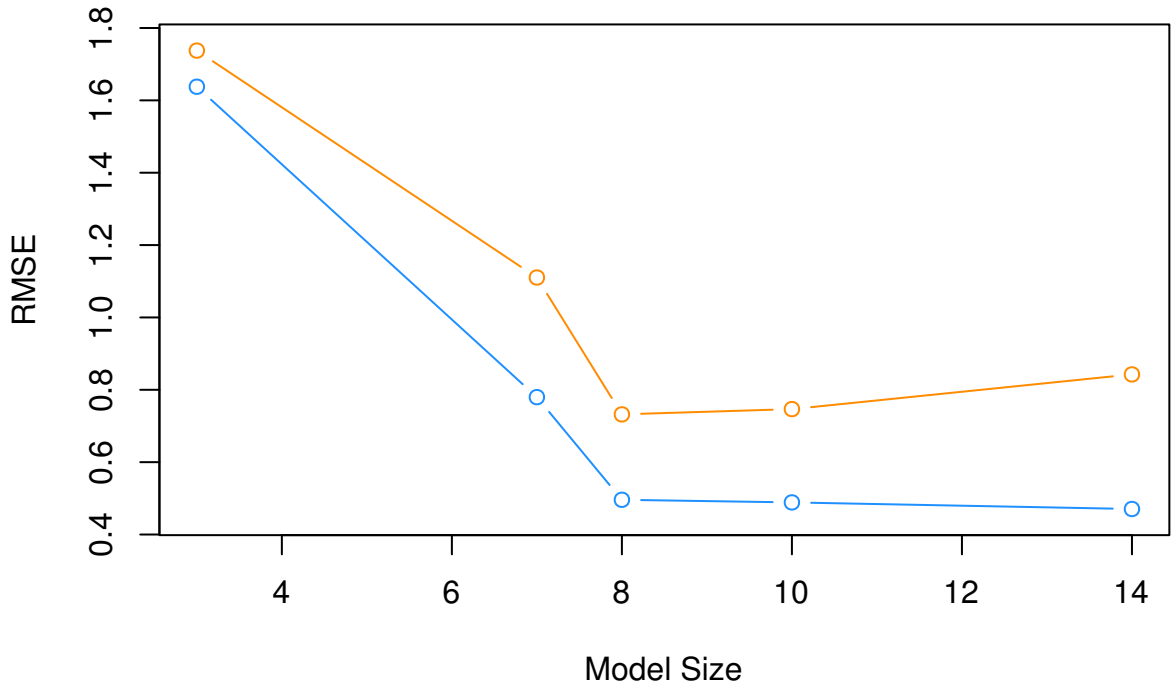
```
model_list = list(fit_1, fit_2, fit_3, fit_4, fit_5)
```

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

Chapter Status: Under Constructions. Main ideas in place but lack narrative. Functional version of much of the code exist but will be cleaned up. Some code and simulation examples need to be expanded.

- TODO: last chapter..
- TODO: recall goal
  - frame around estimating regression function

7.1 Parametric versus Non-Parametric Models

- TODO: How they estimate…

$$f(x) = \mathbb{E}[Y \mid X = x]$$

- TODO: parametric approaches assume form

$$f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p$$

- TODO: non-parametric methods consider locality

$$\hat{f}(x) = \text{average}(\{y_i : x_i = x\})$$

- TODO: since often no points will satisfy that requirement

$$\hat{f}(x) = \text{average}(\{y_i : x_i \text{ equal to (or very close to) x}\})$$

7.2 Local Approaches

- TODO: how do you figure out what is local? what is "close to"?

7.2.1 Neighbors

- example: knn

7.2.2 Neighborhoods

- example: trees

7.3 k-Nearest Neighbors

- TODO: for a concrete example of a non-parametric method...

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

- TODO: how is nearest defined?
  - usually euclidean, but could be any distance
- TODO: implicit minimization (compared to explicit minimization in lm())
  - fitting really just amounts to picking a k, and seeing the training data
- TODO: basic picture
  - for various k's?

7.4 Tuning Parameters versus Model Parameters

- tune (hyper) = how to learn from the data, user specified
  - not specific to non-parametric methods
- model = learned from the data, user specifies how many and form

7.5 KNN in `R`

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

```
set.seed(42)
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

careful 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.

```
knn.reg(train = ?, test = ?, y = ?, k = ?)
```

INPUT

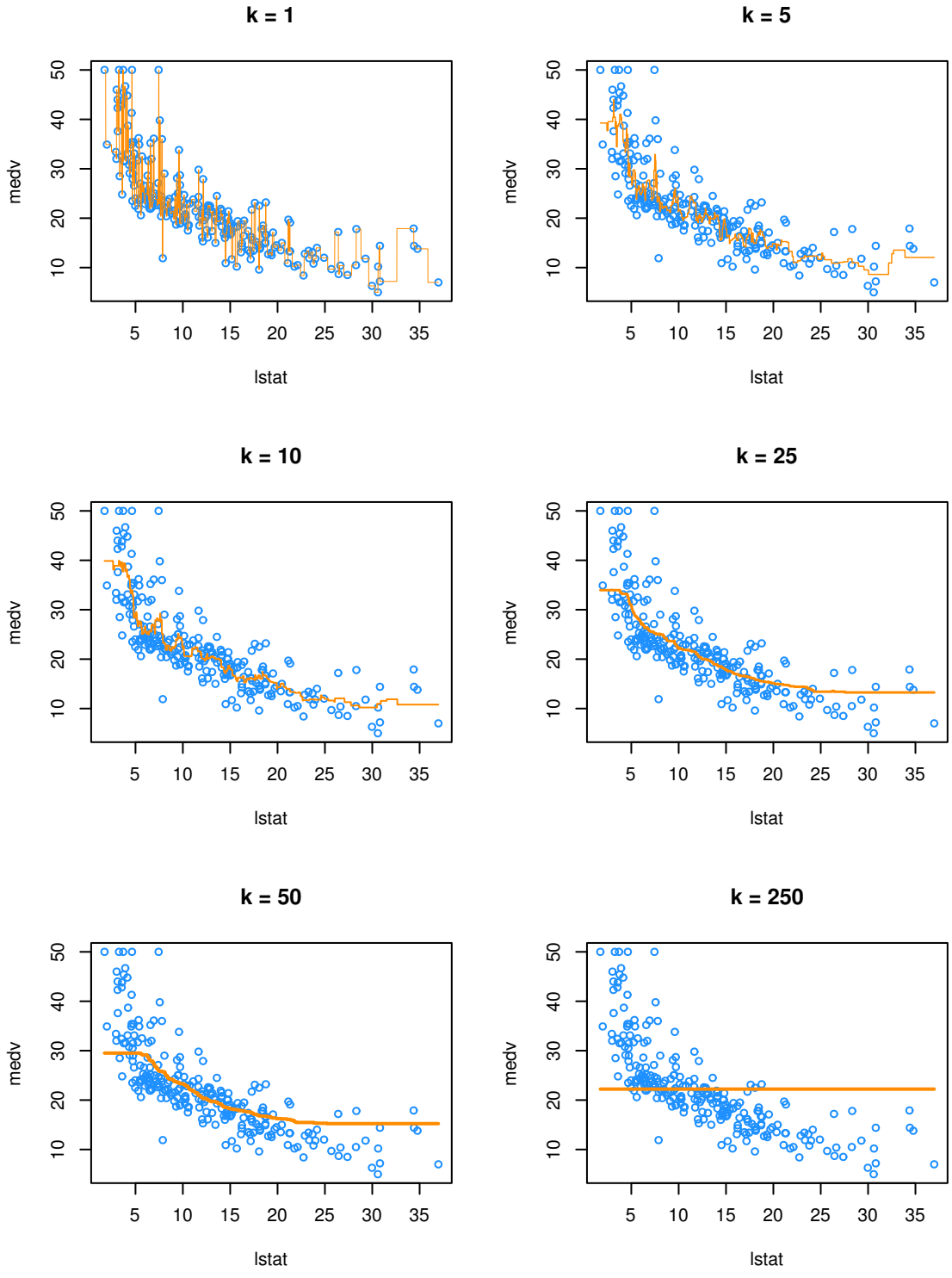
- `train`: the predictors of the training data
- `test`: the predictor values, x, at which we would like to make predictions
- `y`: the response for the training data
- `k`: the number of neighbors to consider

OUTPUT

- the output of `knn.reg()` is exactly $\hat{f}_k(x)$

```
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 a large number of possible values of `lstat`, for different values of `k`. Note that `250` is the total number of observations in this training dataset.



- TODO: Orange "curves" are $\hat{f}_k(x)$ where x are the values we defined in `lstat_grid`. So really a bunch of predictions with interpolated lines, but you can't really tell...

We see that `k = 1` is clearly overfitting, as `k = 1` is a very complex, highly variable model. Conversely, `k = 250` is clearly underfitting the data, as `k = 250` is a very simple, low variance model. In fact, here it is predicting a simple average of all the data at each point.

7.6 Choosing k

- low `k` = very complex model. very wiggly. specifically jagged
- high `k` = very inflexible model. very smooth.
- want: something in the middle which predicts well on unseen data
- that is, want $\hat{f}_k$ to minimize

$$\text{EPE}\left(Y, \hat{f}_k(X)\right) = \mathbb{E}_{X,Y,\mathcal{D}}\left[(Y - \hat{f}_k(X))^2\right]$$

- TODO: Test MSE is an estimate of this. So finding best test RMSE will be our strategy. (Best test RMSE is same as best MSE, but with more understandable units.)

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

```
# define helper function for getting knn.reg predictions
# note: this function is highly specific to this situation and dataset
make_knn_pred = function(k = 1, training, predicting) {
  pred = FNN::knn.reg(train = training["lstat"],
                      test = predicting["lstat"],
                      y = training$medv, k = k)$pred
  act  = predicting$medv
  rmse(predicted = pred, actual = act)
}
```

```
# define values of k to evaluate
k = c(1, 5, 10, 25, 50, 250)
```

```
# get requested train RMSEs
knn_trn_rmse = sapply(k, make_knn_pred,
                      training = trn_boston,
                      predicting = trn_boston)
# get requested test RMSEs
knn_tst_rmse = sapply(k, make_knn_pred,
                      training = trn_boston,
                      predicting = tst_boston)

# determine "best" k
best_k = k[which.min(knn_tst_rmse)]

# find overfitting, underfitting, and "best"" k
fit_status = ifelse(k < best_k, "Over", ifelse(k == best_k, "Best", "Under"))
```

```
# summarize results
knn_results = data.frame(
  k,
  round(knn_trn_rmse, 2),
  round(knn_tst_rmse, 2),
  fit_status
)
colnames(knn_results) = c("k", "Train RMSE", "Test RMSE", "Fit?")
```

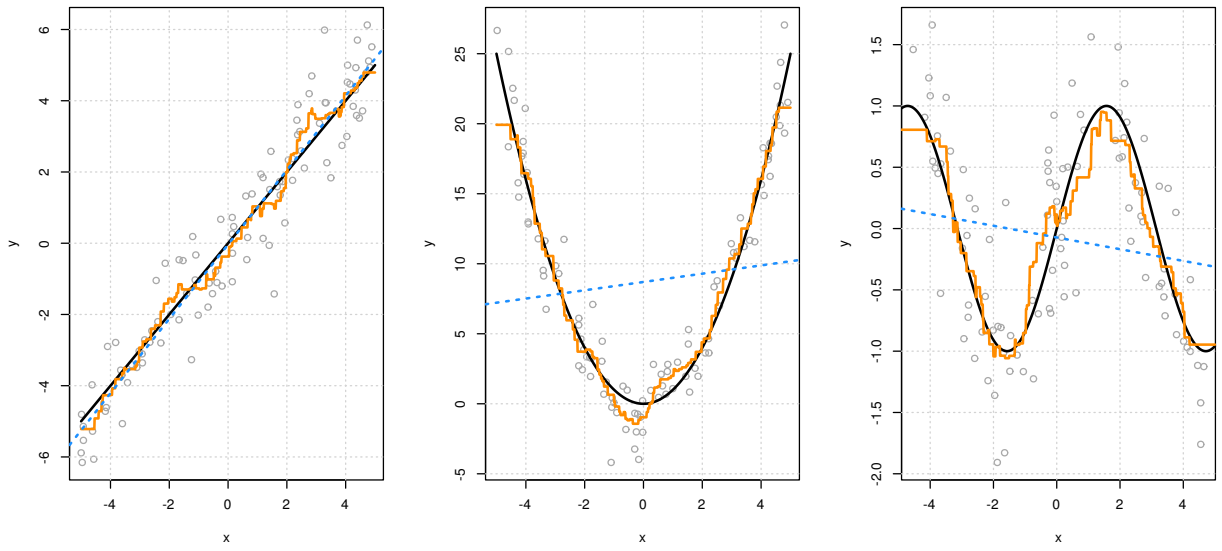
```
# display results
knitr::kable(knn_results, escape = FALSE, booktabs = TRUE)
```

k	Train RMSE	Test RMSE	Fit?
1	2.22	7.50	Over
5	4.30	5.89	Over
10	4.60	5.72	Over
25	4.66	5.71	Best
50	4.99	6.03	Under
250	8.90	9.47	Under

- TODO: What about ties? why isn't k = 1 give 0 training error? There are some non-unique x_i values in the training data. How do we predict when this is the case?

7.7 Linear versus Non-Linear

- TODO; linear relationship example
  - lm() works well
  - knn "automatically" approximates
- TODO: very non-linear example
  - lm() fails badly
    - could work if ...
  - knn "automatically" approximates

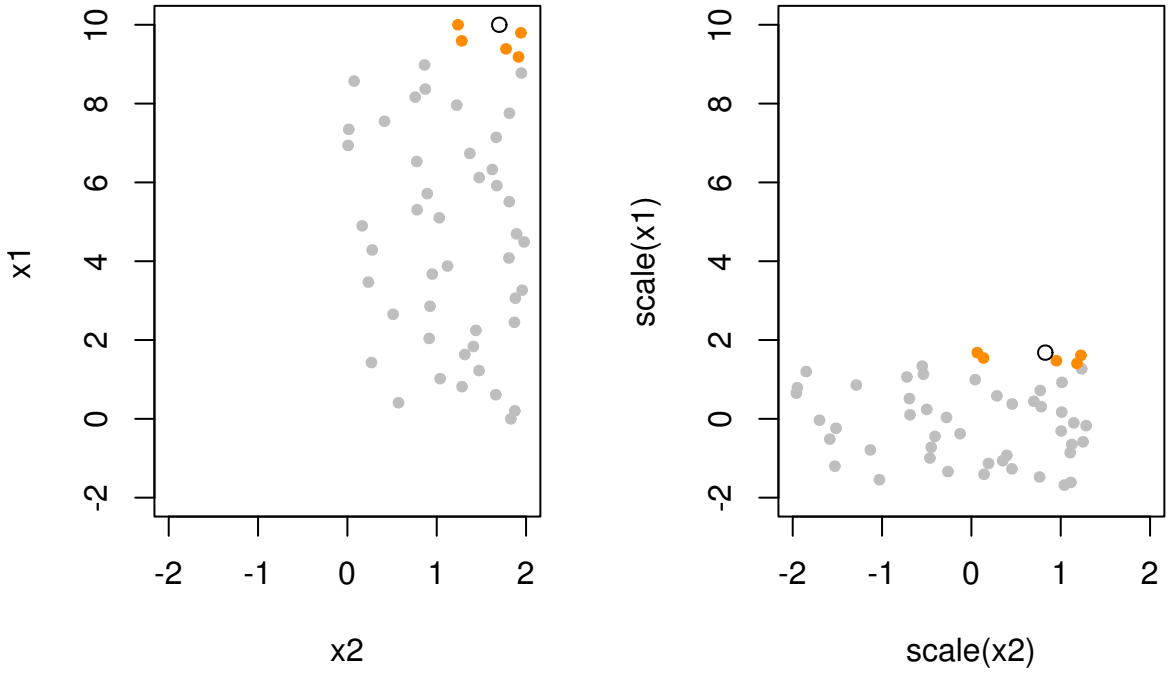


7.8 Scaling Data

- TODO: Sometimes "scale" differentiates between center and scale. `R` function `scale()` does both by default. Outputs variables with mean = 0, var = 1.

```
sim_knn_data = function(n_obs = 50) {
  x1 = seq(0, 10, length.out = n_obs)
  x2 = runif(n = n_obs, min = 0, max = 2)
  x3 = runif(n = n_obs, min = 0, max = 1)
  x4 = runif(n = n_obs, min = 0, max = 5)
  x5 = runif(n = n_obs, min = 0, max = 5)
  y = x1 ^ 2 + rnorm(n = n_obs)
  data.frame(y, x1, x2, x3,x4, x5)
}
```

```
set.seed(42)
knn_data = sim_knn_data()
```



- TODO: How should we scale the test data?
- TODO: Show that linear regression is invariant to scaling. KNN is not. - y = b0 + b1x1 + b2x2 + e - y = b0 + b1x1_ + b2x2_ + e - how are these coefficients related - define how the scaling - RMSE for both, RMSE for both ways KNN

7.9 Curse of Dimensionality

```
set.seed(42)
knn_data_trn = sim_knn_data()
knn_data_tst = sim_knn_data()
```

```
# define helper function for getting knn.reg predictions
# note: this function is highly specific to this situation and dataset
make_knn_pred = function(k = 1, X_trn, X_pred, y_trn, y_pred) {
  pred = FNN::knn.reg(train = scale(X_trn), test = scale(X_pred), y = y_trn, k = k)$pred
  act  = y_pred
  rmse(predicted = pred, actual = act)
}
```

```
# TODO: DRY
cod_train_rmse = c(
  make_knn_pred (k = 5, X_trn = knn_data_trn["x1"],  X_pred = knn_data_trn["x1"],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_trn["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:3], X_pred = knn_data_trn[, 2:3],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_trn["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:4], X_pred = knn_data_trn[, 2:4],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_trn["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:5], X_pred = knn_data_trn[, 2:5],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_trn["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:6], X_pred = knn_data_trn[, 2:6],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_trn["y"]))
```

```
## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA
```

```
# TODO: DRY
cod_test_rmse = c(
  make_knn_pred (k = 5, X_trn = knn_data_trn["x1"],  X_pred = knn_data_tst["x1"],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_tst["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:3], X_pred = knn_data_tst[, 2:3],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_tst["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:4], X_pred = knn_data_tst[, 2:4],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_tst["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:5], X_pred = knn_data_tst[, 2:5],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_tst["y"]),
  make_knn_pred (k = 5, X_trn = knn_data_trn[, 2:6], X_pred = knn_data_tst[, 2:6],
                 y_trn = knn_data_trn["y"], y_pred = knn_data_tst["y"]))
```

```
## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA
```

```
## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA

## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA
```

```
cod_results = data.frame(
  dimension = c(1, 2, 3, 4, 5),
  cod_train_rmse,
  cod_test_rmse
)

colnames(cod_results) = c("$p$, Dimension", "Train RMSE", "Test RMSE")

knitr::kable(cod_results, escape = FALSE, booktabs = TRUE)
```

p, Dimension	Train RMSE	Test RMSE
1	NA	NA
2	NA	NA
3	NA	NA
4	NA	NA
5	NA	NA

- TODO: Local becomes less local.

7.10 Train Time versus Test Time

- TODO: lm vs knn
  - lm: "slow" train, "fast" test
  - knn: "fast" train, "slow" test
  - illustrate with system timings

7.11 Interpretability

- TODO: lm (high) vs knn (low)
  - somewhat generalizes to parametric vs non-parametric

7.12 Data Example

Returning to the `Boston` dataset, we now use all of the available predictors.

```
X_trn_boston = trn_boston[, !names(trn_boston) %in% c("medv")]
X_tst_boston = tst_boston[, !names(tst_boston) %in% c("medv")]
y_trn_boston = trn_boston["medv"]
y_tst_boston = tst_boston["medv"]
```

```
scaled_pred = knn.reg(train = scale(X_trn_boston), test = scale(X_tst_boston),
                      y = y_trn_boston, k = 10)$pred
unscaled_pred = knn.reg(train = X_trn_boston, test = X_tst_boston,
                        y = y_trn_boston, k = 10)$pred

# test rmse
rmse(predicted = scaled_pred, actual = y_tst_boston) # with scaling
```

```
## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA
```

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

```
rmse(predicted = unscaled_pred, actual = y_tst_boston) # without scaling
```

```
## Warning in mean.default((actual - predicted)^2): argument is not numeric or
## logical: returning NA
```

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

Here we see that scaling makes a pretty big difference.

Can you improve this model? Can you find a better k? Can you find a better model by only using some of the predictors?

7.13 rmarkdown

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

```
## [1] "MASS" "FNN"
```

Chapter 8

Bias–Variance Tradeoff

Consider the general regression setup where we are given a random pair $(X, Y) \in \mathbb{R}^p \times \mathbb{R}$. We would like to "predict" Y with some function of X, say, $f(X)$.

To clarify what we mean by "predict," we specify that we would like $f(X)$ to be "close" to Y. To further clarify what we mean by "close," we define the **squared error loss** of estimating Y using $f(X)$.

$$L(Y, f(X)) \triangleq (Y - f(X))^2$$

Now we can clarify the goal of regression, which is to minimize the above loss, on average. We call this the **risk** of estimating Y using $f(X)$.

$$R(Y, f(X)) \triangleq \mathbb{E}[L(Y, f(X))] = \mathbb{E}_{X,Y}[(Y - f(X))^2]$$

Before attempting to minimize the risk, we first re-write the risk after conditioning on X.

$$\mathbb{E}_{X,Y}\left[(Y - f(X))^2\right] = \mathbb{E}_X \mathbb{E}_{Y|X}\left[(Y - f(X))^2 \mid X = x\right]$$

Minimizing the right-hand side is much easier, as it simply amounts to minimizing the inner expectation with respect to $Y \mid X$, essentially minimizing the risk pointwise, for each x.

It turns out, that the risk is minimized by the conditional mean of Y given X,

$$f(x) = \mathbb{E}(Y \mid X = x)$$

which we call the **regression function**.

Note that the choice of squared error loss is somewhat arbitrary. Suppose instead we chose absolute error loss.

$$L(Y, f(X)) \triangleq |Y - f(X)|$$

The risk would then be minimized by the conditional median.

$$f(x) = \text{median}(Y \mid X = x)$$

Despite this possibility, our preference will still be for squared error loss. The reasons for this are numerous, including: historical, ease of optimization, and protecting against large deviations.

Now, given data $\mathcal{D} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}$, our goal becomes finding some $\hat{f}$ that is a good estimate of the regression function f. We'll see that this amounts to minimizing what we call the reducible error.

8.1 Reducible and Irreducible Error

Suppose that we obtain some $\hat{f}$, how well does it estimate f? We define the **expected prediction error** of predicting Y using $\hat{f}(X)$. A good $\hat{f}$ will have a low expected prediction error.

$$\text{EPE}\left(Y, \hat{f}(X)\right) \triangleq \mathbb{E}_{X,Y,\mathcal{D}}\left[\left(Y - \hat{f}(X)\right)^2\right]$$

This expectation is over X, Y, and also $\mathcal{D}$. The estimate $\hat{f}$ is actually random depending on the sampled data $\mathcal{D}$. We could actually write $\hat{f}(X, \mathcal{D})$ to make this dependence explicit, but our notation will become cumbersome enough as it is.

Like before, we'll condition on X. This results in the expected prediction error of predicting Y using $\hat{f}(X)$ when $X = x$.

$$\text{EPE}\left(Y, \hat{f}(x)\right) = \mathbb{E}_{Y|X,\mathcal{D}}\left[\left(Y - \hat{f}(X)\right)^2 \mid X = x\right] = \underbrace{\mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right]}_{\text{reducible error}} + \underbrace{\mathbb{V}_{Y|X}\left[Y \mid X = x\right]}_{\text{irreducible error}}$$

A number of things to note here:

- The expected prediction error is for a random Y given a fixed x and a random $\hat{f}$. As such, the expectation is over $Y \mid X$ and $\mathcal{D}$. Our estimated function $\hat{f}$ is random depending on the sampled data, $\mathcal{D}$, which is used to perform the estimation.
- The expected prediction error of predicting Y using $\hat{f}(X)$ when $X = x$ has been decomposed into two errors:
  - The **reducible error**, which is the expected squared error loss of estimation $f(x)$ using $\hat{f}(x)$ at a fixed point x. The only thing that is random here is $\mathcal{D}$, the data used to obtain $\hat{f}$. (Both f and x are fixed.) We'll often call this reducible error the **mean squared error** of estimating $f(x)$ using $\hat{f}$ at a fixed point x.

    $$\text{MSE}\left(f(x), \hat{f}(x)\right) \triangleq \mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right]$$

  - The **irreducible error**. This is simply the variance of Y given that $X = x$, essentially noise that we do not want to learn. This is also called the **Bayes error**.

As the name suggests, the reducible error is the error that we have some control over. But how do we control this error?

8.2 Bias-Variance Decomposition

After decomposing the expected prediction error into reducible and irreducible error, we can further decompose the reducible error.

Recall the definition of the **bias** of an estimator.

$$\text{bias}(\hat{\theta}) \triangleq \mathbb{E}\left[\hat{\theta}\right] - \theta$$

Also recall the definition of the **variance** of an estimator.

$$\mathbb{V}(\hat{\theta}) = \text{var}(\hat{\theta}) \triangleq \mathbb{E}\left[(\hat{\theta} - \mathbb{E}\left[\hat{\theta}\right])^2\right]$$

Using this, we further decompose the reducible error (mean squared error) into bias squared and variance.

$$\text{MSE}\left(f(x), \hat{f}(x)\right) = \mathbb{E}_{\mathcal{D}}\left[\left(f(x) - \hat{f}(x)\right)^2\right] = \underbrace{\left(f(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2}_{\text{bias}^2\left(\hat{f}(x)\right)} + \underbrace{\mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2\right]}_{\text{var}\left(\hat{f}(x)\right)}$$

This is actually a common fact in estimation theory, but we have stated it here specifically for estimation of some regression function f using $\hat{f}$ at some point x.

$$\text{MSE}\left(f(x), \hat{f}(x)\right) = \text{bias}^2\left(\hat{f}(x)\right) + \text{var}\left(\hat{f}(x)\right)$$

In a perfect world, we would be able to find some $\hat{f}$ which is **unbiased**, that is $\text{bias}\left(\hat{f}(x)\right) = 0$, which also has low variance. In practice, this isn't always possible.

It turns out, there is a **bias-variance tradeoff**. That is, often, the more bias in our estimation, the lesser the variance. Similarly, less variance is often accompanied by more bias. Complex models tend to be unbiased, but highly variable. Simple models are often extremely biased, but have low variance.

In the context of regression, models are biased when:

- Parametric: The form of the model does not incorporate all the necessary variables, or the form of the relationship is too simple. For example, a parametric model assumes a linear relationship, but the true relationship is quadratic.
- Non-parametric: The model provides too much smoothing.

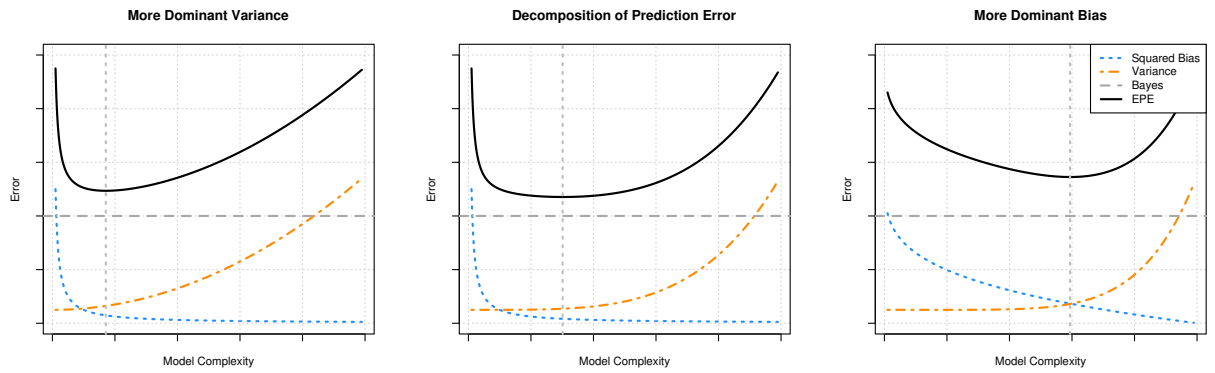
In the context of regression, models are variable when:

- Parametric: The form of the model incorporates too many variables, or the form of the relationship is too complex. For example, a parametric model assumes a cubic relationship, but the true relationship is linear.
- Non-parametric: The model does not provide enough smoothing. It is very, "wiggly."

So for us, to select a model that appropriately balances the tradeoff between bias and variance, and thus minimizes the reducible error, we need to select a model of the appropriate complexity for the data.

Recall that when fitting models, we've seen that train RMSE decreases as model complexity is increasing. (Technically it is non-increasing.) For test RMSE, we expect to see a U-shaped curve. Importantly, test RMSE decreases, until a certain complexity, then begins to increase.

Now we can understand why this is happening. The expected test RMSE is essentially the expected prediction error, which we now known decomposes into (squared) bias, variance, and the irreducible Bayes error. The following plots show three examples of this.



The three plots show three examples of the bias-variance tradeoff. In the left panel, the variance influences the expected prediction error more than the bias. In the right panel, the opposite is true. The middle panel is somewhat neutral. In all cases, the difference between the Bayes error (the horizontal dashed grey line) and the expected prediction error (the solid black curve) is exactly the mean squared error, which is the sum of the squared bias (blue curve) and variance (orange curve). The vertical line indicates the complexity that minimizes the prediction error.

To summarize, if we assume that irreducible error can be written as

$$\mathbb{V}[Y \mid X = x] = \sigma^2$$

then we can write the full decomposition of the expected prediction error of predicting Y using $\hat{f}$ when $X = x$ as

$$\text{EPE}\left(Y, \hat{f}(x)\right) = \underbrace{\text{bias}^2\left(\hat{f}(x)\right) + \text{var}\left(\hat{f}(x)\right)}_{\text{reducible error}} + \sigma^2.$$

As model complexity increases, bias decreases, while variance increases. By understanding the tradeoff between bias and variance, we can manipulate model complexity to find a model that well predict well on unseen observations.

8.3 Simulation

We will illustrate these decompositions, most importantly the bias-variance tradeoff, through simulation. Suppose we would like to train a model to learn the true regression function function $f(x) = x^2$.

```
f = function(x) {
  x ^ 2
}
```

More specifically, we'd like to predict an observation, Y, given that $X = x$ by using $\hat{f}(x)$ where

$$\mathbb{E}[Y \mid X = x] = f(x) = x^2$$

and

$$\mathbb{V}[Y \mid X = x] = \sigma^2.$$

Alternatively, we could write this as

$$Y = f(X) + \epsilon$$

where $\mathbb{E}[\epsilon] = 0$ and $\mathbb{V}[\epsilon] = \sigma^2$. In this formulation, we call $f(X)$ the **signal** and ϵ the **noise**.

To carry out a concrete simulation example, we need to fully specify the data generating process. We do so with the following `R` code.

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

Also note that if you prefer to think of this situation using the $Y = f(X) + \epsilon$ formulation, the following code represents the same data generating process.

```
get_sim_data = function(f, sample_size = 100) {
  x = runif(n = sample_size, min = 0, max = 1)
  eps = rnorm(n = sample_size, mean = 0, sd = 0.75)
  y = f(x) + eps
  data.frame(x, y)
}
```

To completely specify the data generating process, we have made more model assumptions than simply $\mathbb{E}[Y \mid X = x] = x^2$ and $\mathbb{V}[Y \mid X = x] = \sigma^2$. In particular,

- The x_i in $\mathcal{D}$ are sampled from a uniform distribution over $[0, 1]$.
- The x_i and ϵ are independent.
- The y_i in $\mathcal{D}$ are sampled from the conditional normal distribution.

$$Y \mid X \sim N(f(x), \sigma^2)$$

Using this setup, we will generate datasets, $\mathcal{D}$, with a sample size $n = 100$ and fit four models.

$$\begin{aligned}
\texttt{predict(fit0, x)} &= \hat{f}_0(x) = \hat{\beta}_0 \\
\texttt{predict(fit1, x)} &= \hat{f}_1(x) = \hat{\beta}_0 + \hat{\beta}_1 x \\
\texttt{predict(fit2, x)} &= \hat{f}_2(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 \\
\texttt{predict(fit9, x)} &= \hat{f}_9(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \ldots + \hat{\beta}_9 x^9
\end{aligned}$$

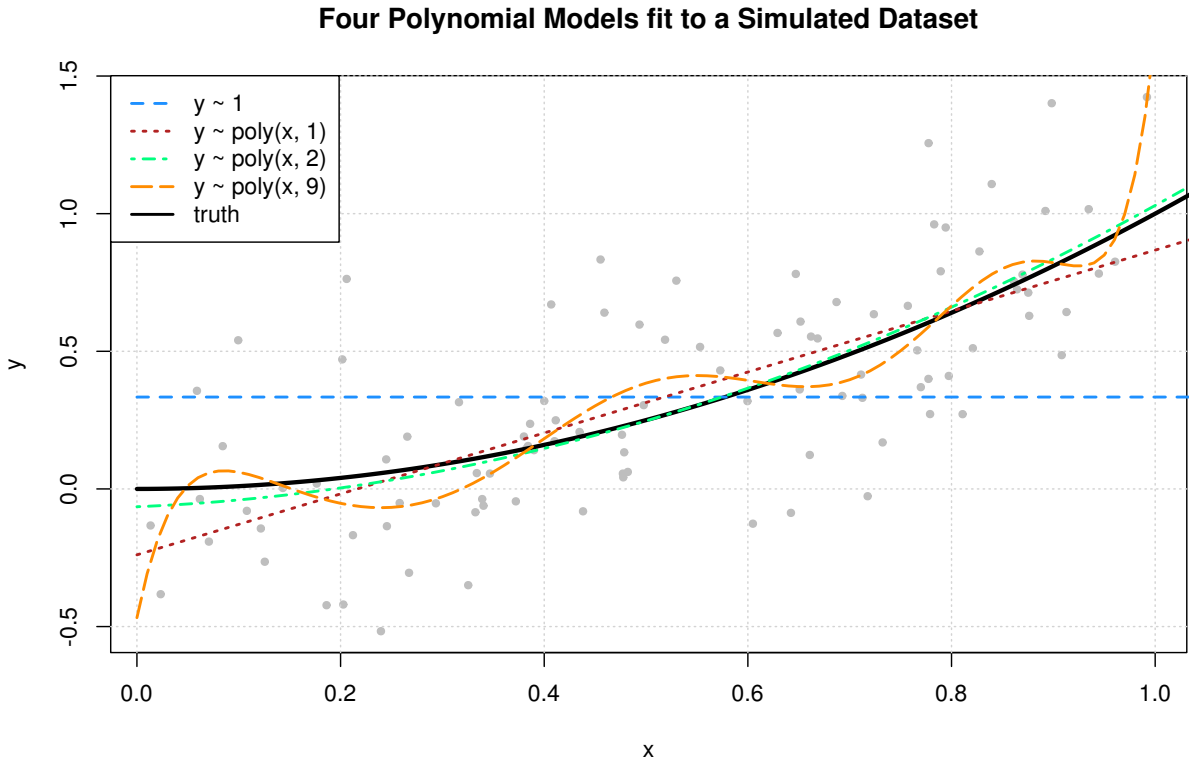
To get a sense of the data and these four models, we generate one simulated dataset, and fit the four models.

```
set.seed(1)
sim_data = get_sim_data(f)
```

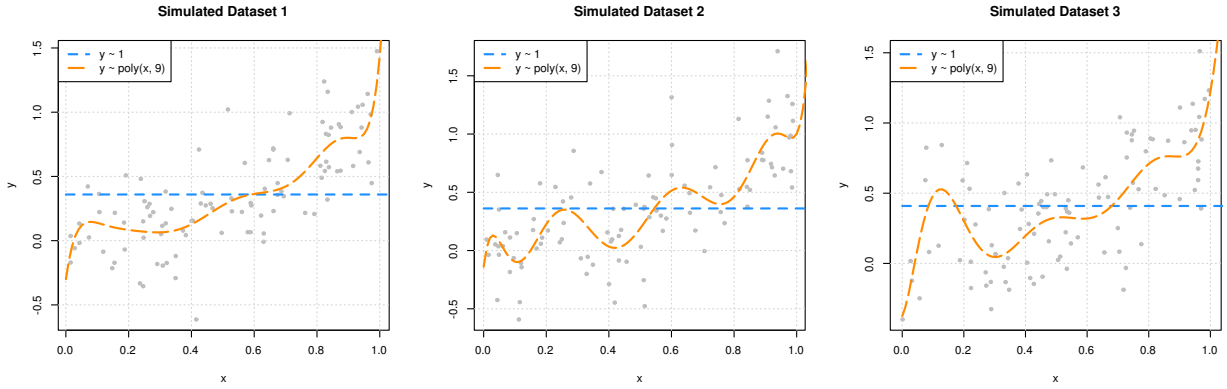
```
fit_0 = lm(y ~ 1,                   data = sim_data)
fit_1 = lm(y ~ poly(x, degree = 1), data = sim_data)
fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
fit_9 = lm(y ~ poly(x, degree = 9), data = sim_data)
```

Note that technically we're being lazy and using orthogonal polynomials, but the fitted values are the same, so this makes no difference for our purposes.

Plotting these four trained models, we see that the zero predictor model does very poorly. The first degree model is reasonable, but we can see that the second degree model fits much better. The ninth degree model seem rather wild.



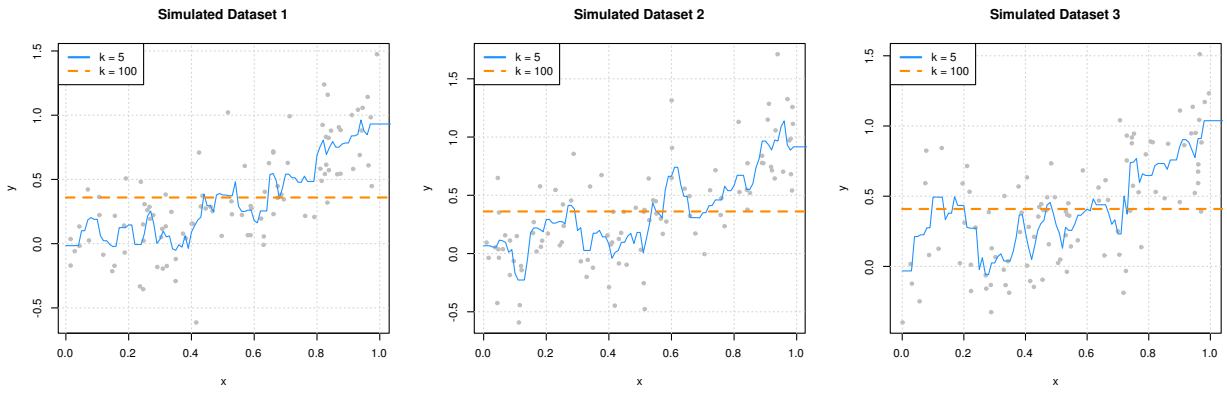
The following three plots were created using three additional simulated datasets. The zero predictor and ninth degree polynomial were fit to each.



This plot should make clear the difference between the bias and variance of these two models. The zero predictor model is clearly wrong, that is, biased, but nearly the same for each of the datasets, since it has very low variance.

While the ninth degree model doesn't appear to be correct for any of these three simulations, we'll see that on average it is, and thus is performing unbiased estimation. These plots do however clearly illustrate that the ninth degree polynomial is extremely variable. Each dataset results in a very different fitted model. Correct on average isn't the only goal we're after, since in practice, we'll only have a single dataset. This is why we'd also like our models to exhibit low variance.

We could have also fit k-nearest neighbors models to these three datasets.



Here we see that when $k = 100$ we have a biased model with very low variance. (It's actually the same as the 0 predictor linear model.) When $k = 5$, we again have a highly variable model.

These two sets of plots reinforce our intuition about the bias-variance tradeoff. Complex models (ninth degree polynomial and $k = 5$) are highly variable, and often unbiased. Simple models (zero predictor linear model and $k = 100$) are very biased, but have extremely low variance.

We will now complete a simulation study to understand the relationship between the bias, variance, and mean squared error for the estimates for $f(x)$ given by these four models at the point $x = 0.90$. We use simulation to complete this task, as performing the analytical calculations would prove to be rather tedious and difficult.

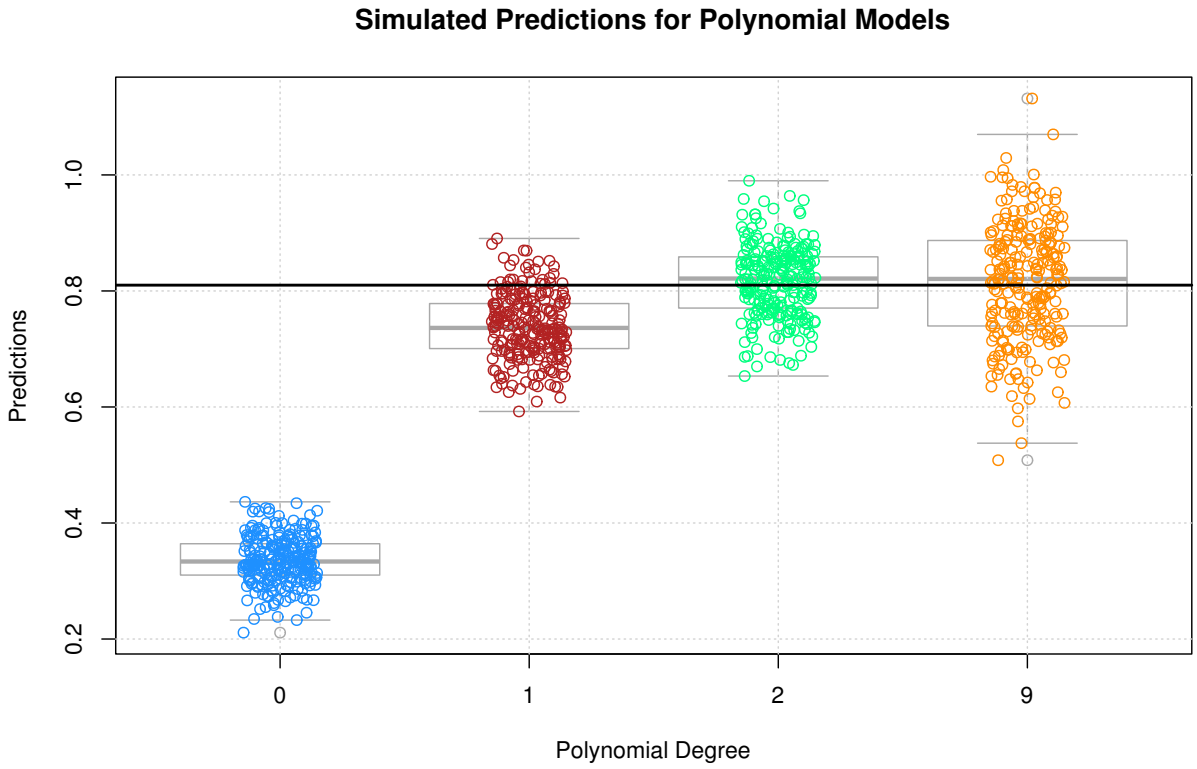
```
set.seed(1)
n_sims = 250
n_models = 4
x = data.frame(x = 0.90) # fixed point at which we make predictions
predictions = matrix(0, nrow = n_sims, ncol = n_models)
```

```
for (sim in 1:n_sims) {

  # simulate new, random, training data
  # this is the only random portion of the bias, var, and mse calculations
  # this allows us to calculate the expectation over D
  sim_data = get_sim_data(f)

  # fit models
  fit_0 = lm(y ~ 1,                   data = sim_data)
  fit_1 = lm(y ~ poly(x, degree = 1), data = sim_data)
  fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
  fit_9 = lm(y ~ poly(x, degree = 9), data = sim_data)

  # get predictions
  predictions[sim, 1] = predict(fit_0, x)
  predictions[sim, 2] = predict(fit_1, x)
  predictions[sim, 3] = predict(fit_2, x)
  predictions[sim, 4] = predict(fit_9, x)
}
```

Note that this is one of many ways we could have accomplished this task using R. For example we could have used a combination of `replicate()` and `*apply()` functions. Alternatively, we could have used a `tidyverse` approach, which likely would have used some combination of `dplyr`, `tidyr`, and `purrr`.

Our approach, which would be considered a `base R` approach, was chosen to make it as clear as possible what is being done. The `tidyverse` approach is rapidly gaining popularity in the `R` community, but might make it more difficult to see what is happening here, unless you are already familiar with that approach.

Also of note, while it may seem like the output stored in `predictions` would meet the definition of tidy data given by Hadley Wickham since each row represents a simulation, it actually falls slightly short. For our data to be tidy, a row should store the simulation number, the model, and the resulting prediction. We've actually already aggregated one level above this. Our observational unit is a simulation (with four predictions), but for tidy data, it should be a single prediction. This may be revised by the author later when there are more examples of how to do this from the `R` community.



The above plot shows the predictions for each of the 250 simulations of each of the four models of different polynomial degrees. The truth, $f(x = 0.90) = (0.9)^2 = 0.81$, is given by the solid black horizontal line.

Two things are immediately clear:

- As complexity *increases*, **bias decreases**. (The mean of a model's predictions is closer to the truth.)
- As complexity *increases*, **variance increases**. (The variance about the mean of a model's predictions increases.)

The goal of this simulation study is to show that the following holds true for each of the four models.

$$\text{MSE}\left(f(0.90), \hat{f}_k(0.90)\right) = \underbrace{\left(\mathbb{E}\left[\hat{f}_k(0.90)\right] - f(0.90)\right)^2}_{\text{bias}^2\left(\hat{f}_k(0.90)\right)} + \underbrace{\mathbb{E}\left[\left(\hat{f}_k(0.90) - \mathbb{E}\left[\hat{f}_k(0.90)\right]\right)^2\right]}_{\text{var}\left(\hat{f}_k(0.90)\right)}$$

We'll use the empirical results of our simulations to estimate these quantities. (Yes, we're using estimation to justify facts about estimation.) Note that we've actually used a rather small number of simulations. In

practice we should use more, but for the sake of computation time, we've performed just enough simulations to obtain the desired results. (Since we're estimating estimation, the bigger the sample size, the better.)

To estimate the mean squared error of our predictions, we'll use

$$\widehat{\text{MSE}}\left(f(0.90), \hat{f}_k(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(f(0.90) - \hat{f}_k(0.90)\right)^2$$

We also write an accompanying `R` function.

```
get_mse = function(truth, estimate) {
  mean((estimate - truth) ^ 2)
}
```

Similarly, for the bias of our predictions we use,

$$\widehat{\text{bias}}\left(\hat{f}(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(\hat{f}_k(0.90)\right) - f(0.90)$$

And again, we write an accompanying `R` function.

```
get_bias = function(estimate, truth) {
  mean(estimate) - truth
}
```

Lastly, for the variance of our predictions we have

$$\widehat{\text{var}}\left(\hat{f}(0.90)\right) = \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \left(\hat{f}_k(0.90) - \frac{1}{n_{\texttt{sims}}} \sum_{i=1}^{n_{\texttt{sims}}} \hat{f}_k(0.90)\right)^2$$

While there is already `R` function for variance, the following is more appropriate in this situation.

```
get_var = function(estimate) {
  mean((estimate - mean(estimate)) ^ 2)
}
```

To quickly obtain these results for each of the four models, we utilize the `apply()` function.

```
bias = apply(predictions, 2, get_bias, truth = f(x = 0.90))
variance = apply(predictions, 2, get_var)
mse = apply(predictions, 2, get_mse, truth = f(x = 0.90))
```

We summarize these results in the following table.

Degree	Mean Squared Error	Bias Squared	Variance
0	0.22643	0.22476	0.00167
1	0.00829	0.00508	0.00322
2	0.00387	0.00005	0.00381
9	0.01019	0.00002	0.01017

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** 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 increasing 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) < 0
```

```
##     1     2     9
##  TRUE  TRUE FALSE
```

The models with polynomial degrees 2 and 9 are both essentially unbiased. We see some bias here as a result of using simulation. If we increased the number of simulations, we would see both biases go down. Since they are both unbiased, the model with degree 2 outperforms the model with degree 9 due to its smaller variance.

Models with degree 0 and 1 are biased because they assume the wrong form of the regression function. While the degree 9 model does this as well, it does include all the necessary polynomial degrees.

$$\hat{f}_9(x) = \hat{\beta}_0 + \hat{\beta}_1 x + \hat{\beta}_2 x^2 + \ldots + \hat{\beta}_9 x^9$$

Then, since least squares estimation is unbiased, importantly,

$$\mathbb{E}[\hat{\beta}_d] = \beta_d = 0$$

for $d = 3, 4, \ldots 9$, we have

$$\mathbb{E}\left[\hat{f}_9(x)\right] = \beta_0 + \beta_1 x + \beta_2 x^2$$

Now we can finally verify the bias-variance decomposition.

```
bias ^ 2 + variance == mse
```

```
##     0     1     2     9
## FALSE FALSE FALSE  TRUE
```

But wait, this says it isn't true, except for the degree 9 model? It turns out, this is simply a computational issue. If we allow for some very small error tolerance, we see that the bias-variance decomposition is indeed true for predictions from these for models.

```
all.equal(bias ^ 2 + variance, mse)
```

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

See `?all.equal()` for details.

So far, we've focused our efforts on looking at the mean squared error of estimating $f(0.90)$ using $\hat{f}(0.90)$. We could also look at the expected prediction error of using $\hat{f}(X)$ when $X = 0.90$ to estimate Y.

$$\text{EPE}\left(Y, \hat{f}_k(0.90)\right) = \mathbb{E}_{Y|X,\mathcal{D}}\left[\left(Y - \hat{f}_k(X)\right)^2 \mid X = 0.90\right]$$

We can estimate this quantity for each of the four models using the simulation study we already performed.

```
get_epe = function(realized, estimate) {
  mean((realized - estimate) ^ 2)
}
```

```
y = rnorm(n = nrow(predictions), mean = f(x = 0.9), sd = 0.3)
epe = apply(predictions, 2, get_epe, realized = y)
epe
```

```
##         0         1         2         9
## 0.3180470 0.1104055 0.1095955 0.1205570
```

What about the unconditional expected prediction error. That is, for any X, not just 0.90. Specifically, the expected prediction error of estimating Y using $\hat{f}(X)$. The following (new) simulation study provides an estimate of

$$\text{EPE}\left(Y, \hat{f}_k(X)\right) = \mathbb{E}_{X,Y,\mathcal{D}}\left[\left(Y - \hat{f}_k(X)\right)^2\right]$$

for the quadratic model, that is $k = 2$ as we have defined k.

```
set.seed(1)
n_sims = 1000
X = runif(n = n_sims, min = 0, max = 1)
Y = rnorm(n = n_sims, mean = f(X), sd = 0.3)

f_hat_X = rep(0, length(X))

for (i in seq_along(X)) {
  sim_data = get_sim_data(f)
  fit_2 = lm(y ~ poly(x, degree = 2), data = sim_data)
  f_hat_X[i] = predict(fit_2, newdata = data.frame(x = X[i]))
}

mean((Y - f_hat_X) ^ 2)
```

```
## [1] 0.09997319
```

Note that in practice, we should use many more simulations in this study.

8.4 Estimating Expected Prediction Error

While previously, we only decomposed the expected prediction error conditionally, a similar argument holds unconditionally.

Assuming

$$\mathbb{V}[Y \mid X = x] = \sigma^2.$$

we have

$$\text{EPE}\left(Y, \hat{f}(X)\right) = \mathbb{E}_{X,Y,\mathcal{D}}\left[(Y - \hat{f}(X))^2\right] = \underbrace{\mathbb{E}_X\left[\text{bias}^2\left(\hat{f}(X)\right)\right] + \mathbb{E}_X\left[\text{var}\left(\hat{f}(X)\right)\right]}_{\text{reducible error}} + \sigma^2$$

Lastly, we note that if

$$\mathcal{D} = \mathcal{D}_{\texttt{trn}} \cup \mathcal{D}_{\texttt{tst}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i = 1, 2, \ldots n$$

where

$$\mathcal{D}_{\texttt{trn}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i \in \texttt{trn}$$

and

$$\mathcal{D}_{\texttt{tst}} = (x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, \ i \in \texttt{tst}$$

Then, if we use $\mathcal{D}_{\texttt{trn}}$ to fit (train) a model, we can use the test mean squared error

$$\sum_{i \in \texttt{tst}} \left(y_i - \hat{f}(x_i)\right)^2$$

as an estimate of

$$\mathbb{E}_{X,Y,\mathcal{D}}\left[(Y - \hat{f}(X))^2\right]$$

the expected prediction error. (In practice we prefer RMSE to MSE for comparing models and reporting because of the units.)

How good is this estimate? Well, if $\mathcal{D}$ is a random sample from (X, Y), and `tst` are randomly sampled observations randomly sampled from $i = 1, 2, \ldots, n$, then it is a reasonable estimate. However, it is rather variable due to the randomness of selecting the observations for the test set. How variable? It turns out, pretty variable. While it's a justified estimate, eventually we'll introduce cross-validation as a procedure better suited to performing this estimation to select a model.

8.5 `rmarkdown`

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1.

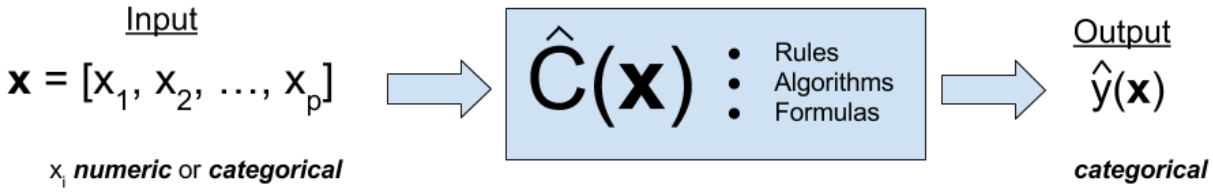
Part III

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.*



That is, the classifier $\hat{C}(x)$ returns the predicted category $\hat{y}(X)$.

$$\hat{y}(x) = \hat{C}(x)$$

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
##    <fct>   <fct>     <dbl>  <dbl>
##  1 No      No         730. 44362.
##  2 No      Yes        817. 12106.
##  3 No      No        1074. 31767.
##  4 No      No         529. 35704.
##  5 No      No         786. 38463.
##  6 No      Yes        920.  7492.
##  7 No      No         826. 24905.
##  8 No      Yes        809. 17600.
##  9 No      No        1161. 37469.
## 10 No      No           0  29275.
## # ... 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)
default_idx   = sample(nrow(Default), 5000)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

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.

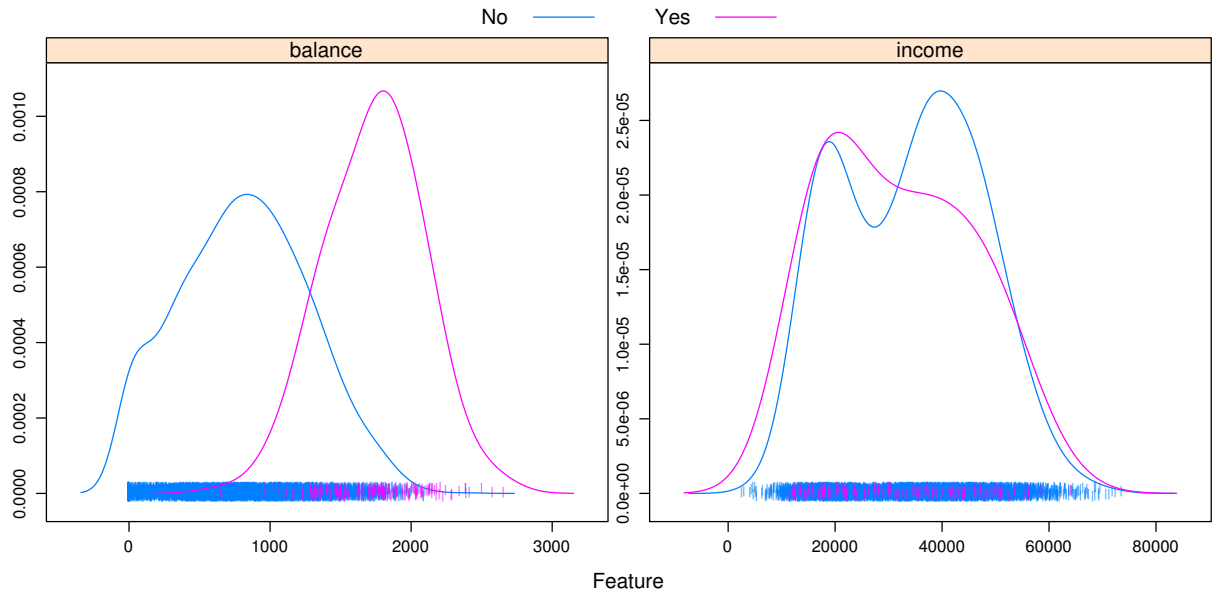
```
library(caret)
```

A density plot can often suggest a simple split based on a numeric predictor. Essentially this plot graphs a density estimate

$$\hat{f}_{X_i}(x_i \mid Y = k)$$

for each numeric predictor x_i and each category k of the response y.

```
featurePlot(x = default_trn[, c("balance", "income")],
            y = default_trn$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))
```

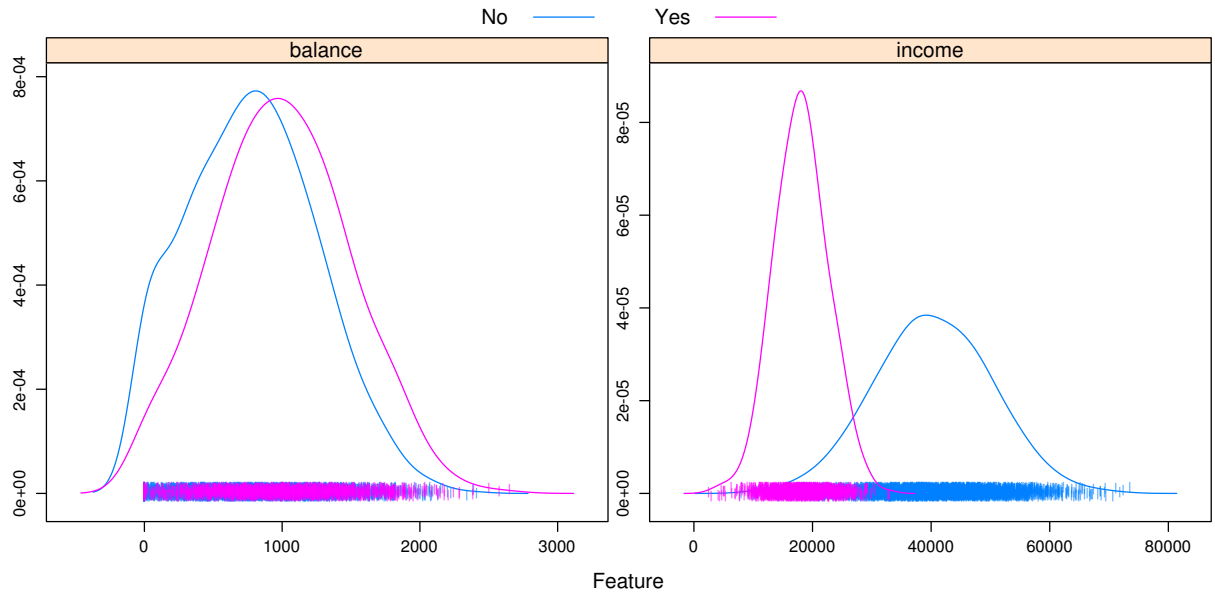


Some notes about the arguments to this function:

- `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.

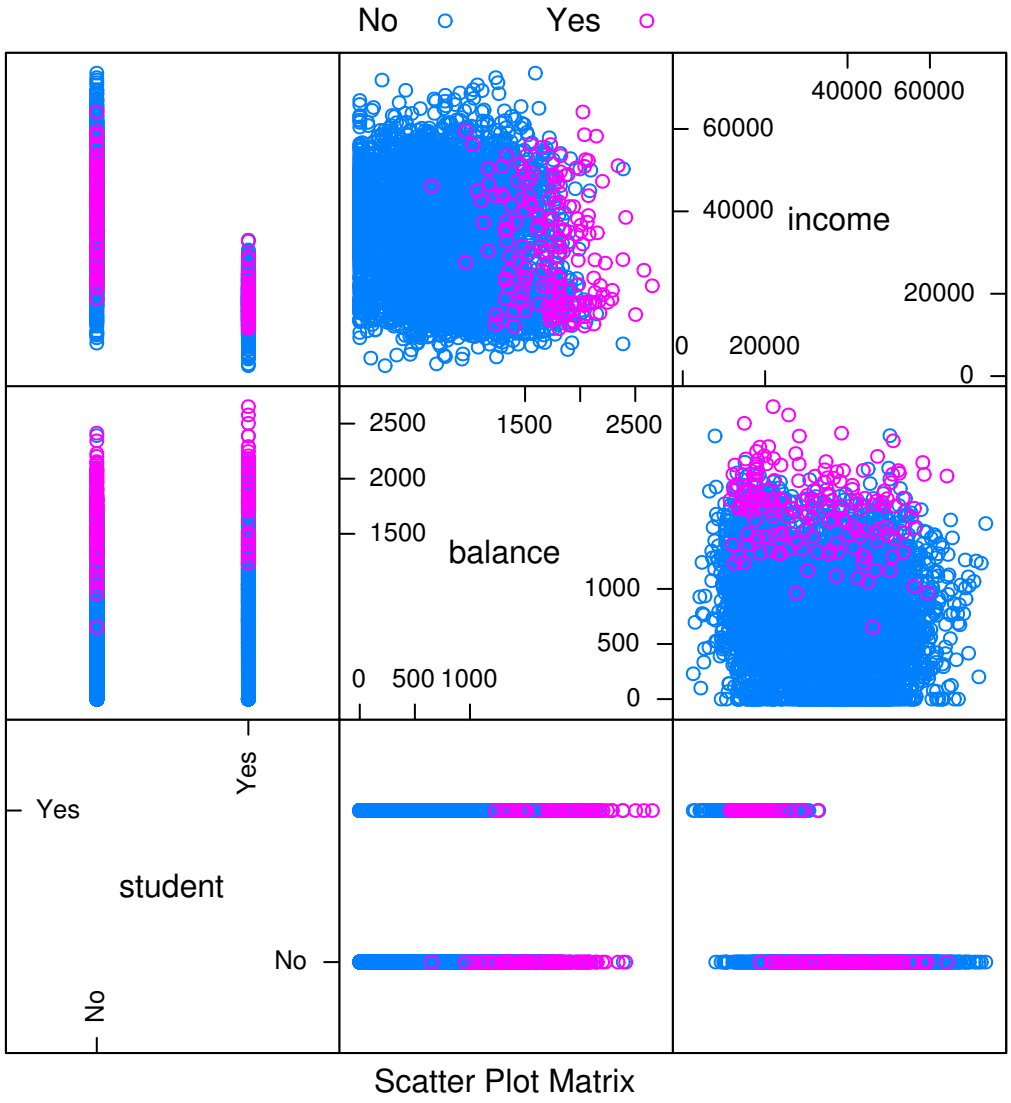
It seems that the income variable by itself is not particularly useful. However, there seems to be a big difference in default status at a `balance` of about 1400. We will use this information shortly.

```
featurePlot(x = default_trn[, c("balance", "income")],
            y = default_trn$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))
```



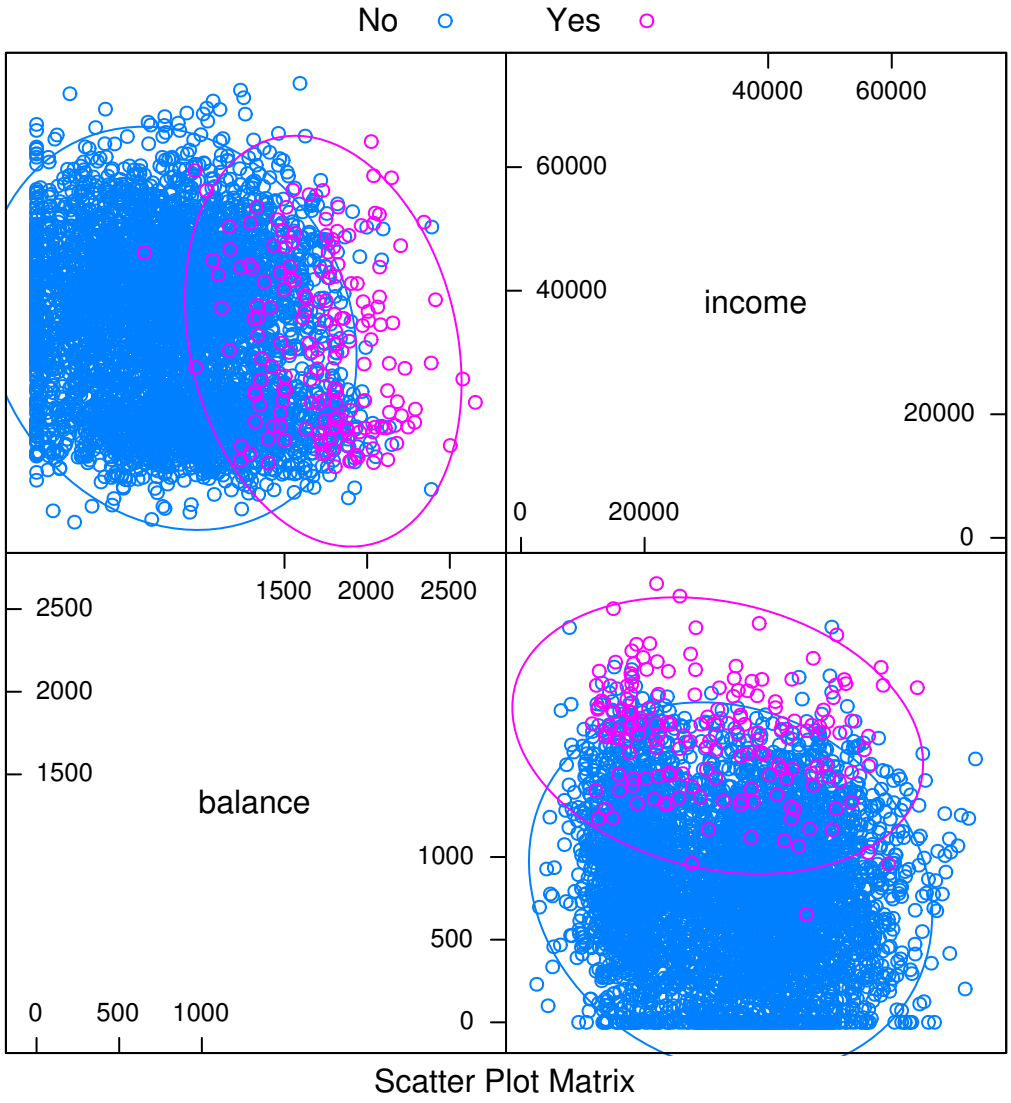
Above, we create a similar plot, except with `student` as the response. We see that students often carry a slightly larger balance, and have far lower income. This will be useful to know when making more complicated classifiers.

```
featurePlot(x = default_trn[, c("student", "balance", "income")],
            y = default_trn$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 = default_trn[, c("balance", "income")],
            y = default_trn$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.

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

Based on the first plot, we believe we can use `balance` to create a reasonable classifier. In particular,

$$\hat{C}(\texttt{balance}) = \begin{cases} \text{Yes} & \texttt{balance} > 1400 \\ \text{No} & \texttt{balance} \leq 1400 \end{cases}$$

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.

```
default_trn_pred = simple_class(x = default_trn$balance,
                                boundary = 1400, above = "Yes", below = "No")
default_tst_pred = simple_class(x = default_tst$balance,
                                boundary = 1400, above = "Yes", below = "No")
head(default_tst_pred, n = 10)
```

```
##  [1] "No" "No" "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.

```
(trn_tab = table(predicted = default_trn_pred, actual = default_trn$default))
```

```
##          actual
## predicted   No  Yes
##       No  4319   29
##       Yes  513  139
```

```
(tst_tab = table(predicted = default_tst_pred, actual = default_tst$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.

		Actual	
		False (0)	**True** (1)
Predicted	**False** (0)	True Negative (**TN**)	False Negative (**FN**)
	True (1)	False Positive (**FP**)	True Positive (**TP**)

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."

```
trn_con_mat  = confusionMatrix(trn_tab, positive = "Yes")
(tst_con_mat = confusionMatrix(tst_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 error rate**.

$$\text{err}(\hat{C}, \text{Data}) = \frac{1}{n} \sum_{i=1}^{n} I(y_i \neq \hat{C}(x_i))$$

Here, I is an indicator function, so we are essentially calculating the proportion of predicted classes that match the true class.

$$I(y_i \neq \hat{C}(x)) = \begin{cases} 1 & y_i \neq \hat{C}(x) \\ 0 & y_i = \hat{C}(x) \end{cases}$$

It is also common to discuss the **accuracy**, which is simply one minus the error.

Like regression, we often split the data, and then consider Train (Classification) Error and Test (Classification) Error will be used as a measure of how well a classifier will work on unseen future data.

$$\text{err}_{\texttt{trn}}(\hat{C}, \text{Train Data}) = \frac{1}{n_{\texttt{trn}}} \sum_{i \in \texttt{trn}} I(y_i \neq \hat{C}(x_i))$$

$$\text{err}_{\texttt{tst}}(\hat{C}, \text{Test Data}) = \frac{1}{n_{\texttt{tst}}} \sum_{i \in \texttt{tst}} I(y_i \neq \hat{C}(x_i))$$

Accuracy values can be found by calling `confusionMatrix()`, or, if stored, can be accessed directly. Here, we use them to obtain error rates.

```
1 - trn_con_mat$overall["Accuracy"]
```

```
## Accuracy
##   0.1084
```

```
1 - tst_con_mat$overall["Accuracy"]
```

```
## Accuracy
##   0.0994
```

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**.

$$\text{Sens} = \text{True Positive Rate} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

```
tst_con_mat$byClass["Sensitivity"]
```

```
## Sensitivity
##   0.8606061
```

$$\text{Spec} = \text{True Negative Rate} = \frac{\text{TN}}{\text{N}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

```
tst_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.

$$\text{Prev} = \frac{\text{P}}{\text{Total Obs}} = \frac{\text{TP} + \text{FN}}{\text{Total Obs}}$$

```
trn_con_mat$byClass["Prevalence"]
```

```
## Prevalence
##     0.0336
```

```
tst_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`.

$$\hat{C}(\texttt{balance}) = \begin{cases} \text{No} & \texttt{balance} > 1400 \\ \text{No} & \texttt{balance} \leq 1400 \end{cases}$$

This classifier simply classifies all observations as negative cases.

```
pred_all_no = simple_class(default_tst$balance,
                           boundary = 1400, above = "No", below = "No")
table(predicted = pred_all_no, actual = default_tst$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 error rates directly. To do so, we write a function.

```
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

```
calc_class_err(actual = default_tst$default,
               predicted = pred_all_no)
```

```
## [1] 0.033
```

Here we see that the error rate is exactly the prevelance of the minority class.

```
table(default_tst$default) / length(default_tst$default)
```

```
##
##    No   Yes
## 0.967 0.033
```

This classifier does better than the previous. But the point is, in reality, to create a good classifier, we should obtain a test error better than 0.033, which is obtained by simply manipulating the prevalences. Next chapter, we'll introduce much better classifiers which should have no problem accomplishing this task.

9.4 rmarkdown

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

```
## [1] "ellipse" "caret"   "ggplot2" "lattice" "tibble"  "ISLR"
```

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
##    <fct>   <fct>     <dbl>  <dbl>
##  1 No      No         730. 44362.
##  2 No      Yes        817. 12106.
##  3 No      No        1074. 31767.
##  4 No      No         529. 35704.
##  5 No      No         786. 38463.
##  6 No      Yes        920.  7492.
##  7 No      No         826. 24905.
##  8 No      Yes        809. 17600.
##  9 No      No        1161. 37469.
## 10 No      No           0  29275.
## # ... with 9,990 more rows
```

We also repeat the test-train split from the previous chapter.

```
set.seed(42)
default_idx = sample(nrow(Default), 5000)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

10.1 Linear Regression

Before moving on to logistic regression, why not plain, old, linear regression?

```
default_trn_lm = default_trn
default_tst_lm = default_tst
```

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_trn_lm$default = as.numeric(default_trn_lm$default) - 1
default_tst_lm$default = as.numeric(default_tst_lm$default) - 1
```

Why would we think this should work? Recall that,

$$\hat{\mathbb{E}}[Y \mid X = x] = X\hat{\beta}.$$

Since Y is limited to values of 0 and 1, we have

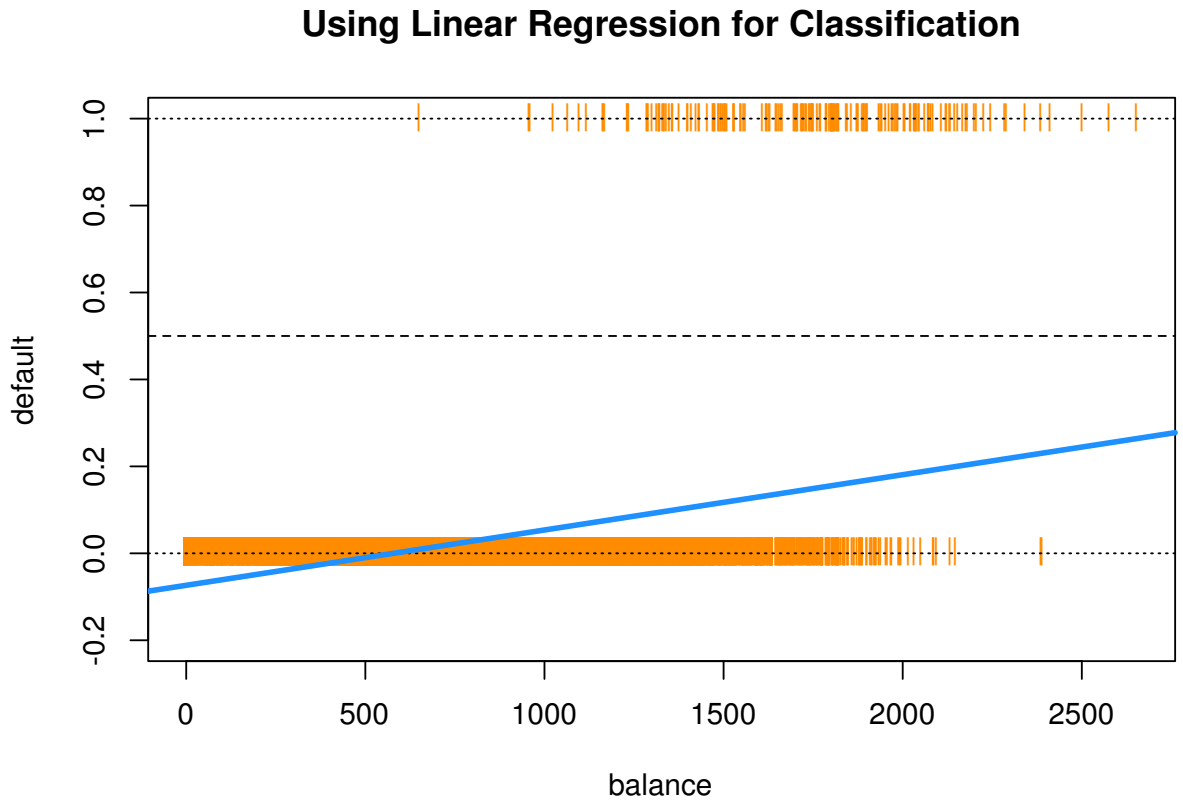
$$\mathbb{E}[Y \mid X = x] = P(Y = 1 \mid X = x).$$

It would then seem reasonable that $\mathbf{X}\hat{\beta}$ 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_trn_lm)
```

Everything seems to be working, until we plot the results.

```
plot(default ~ balance, data = default_trn_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:

$$C^B(x) = \underset{g}{\text{argmax}} \; P(Y = g \mid X = x)$$

So, in the binary classification problem, we will use predicted probabilities

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

and

$$\hat{P}(Y = 0 \mid X = x)$$

and then classify to the larger of the two. We actually only need to consider a single probability, usually $\hat{P}(Y = 1 \mid X = x)$. Since we use it so often, we give it the shorthand notation, $\hat{p}(x)$. Then the classifier is written,

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

This classifier is essentially estimating the Bayes Classifier, thus, is seeking to minimize classification errors.

10.3 Logistic Regression with `glm()`

To better estimate the probability

$$p(x) = P(Y = 1 \mid X = x)$$

we turn to logistic regression. The model is written

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Rearranging, we see the probabilities can be written as

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)}} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)$$

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! (This is actualy a particular sigmoid function called the logistic function, but since it is by far the most popular sigmoid function, often sigmoid function is used to refer to the logistic function)

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

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_trn, 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

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

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

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

where

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p.$$

The commented line, which would give the same results, is performing

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

where

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x).$$

Once we have classifications, we can calculate metrics such as the trainging classification error rate.

```
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

```
calc_class_err(actual = default_trn$default, predicted = model_glm_pred)
```

```
## [1] 0.0278
```

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_trn$default)
library(caret)
train_con_mat = confusionMatrix(train_tab, positive = "Yes")
c(train_con_mat$overall["Accuracy"],
  train_con_mat$byClass["Sensitivity"],
  train_con_mat$byClass["Specificity"])
```

```
##    Accuracy Sensitivity Specificity
##   0.9722000   0.2738095   0.9964818
```

We could also write a custom function for the error for use with trained logist regression models.

```
get_logistic_error = 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)
  calc_class_err(actual = data[, res], predicted = preds)
}
```

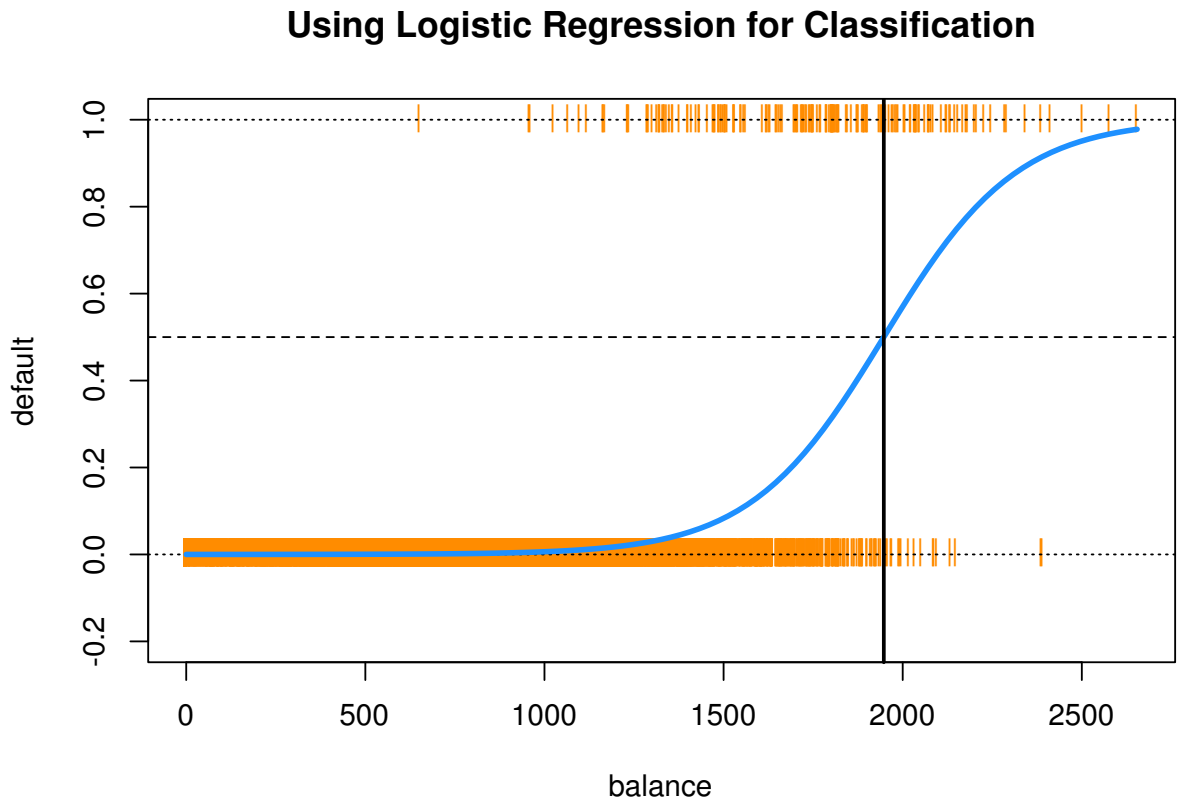
This function will be useful later when calculating train and test errors for several models at the same time.

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

```
## [1] 0.0278
```

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_trn_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,

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

- 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

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x) = 0.5$$

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_trn$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 or remove complexity from logistic regressions.

```
model_1 = glm(default ~ 1, data = default_trn, family = "binomial")
model_2 = glm(default ~ ., data = default_trn, family = "binomial")
model_3 = glm(default ~ . ^ 2 + I(balance ^ 2),
              data = default_trn, 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_errors = sapply(model_list, get_logistic_error, data = default_trn,
                      res = "default", pos = "Yes", neg = "No", cut = 0.5)
test_errors  = sapply(model_list, get_logistic_error, data = default_tst,
                      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_errors)
```

```
## [1] -0.0058 -0.0002
```

```
diff(test_errors)
```

```
## [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_trn, family = "binomial")
```

We write a function which allows use to make predictions based on different probability cutoffs.

```
get_logistic_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)
}
```

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

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_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.1)
test_pred_50 = get_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.5)
test_pred_90 = get_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.9)
```

Now we evaluate accuracy, sensitivity, and specificity for these classifiers.

```
test_tab_10 = table(predicted = test_pred_10, actual = default_tst$default)
test_tab_50 = table(predicted = test_pred_50, actual = default_tst$default)
test_tab_90 = table(predicted = test_pred_90, actual = default_tst$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
```

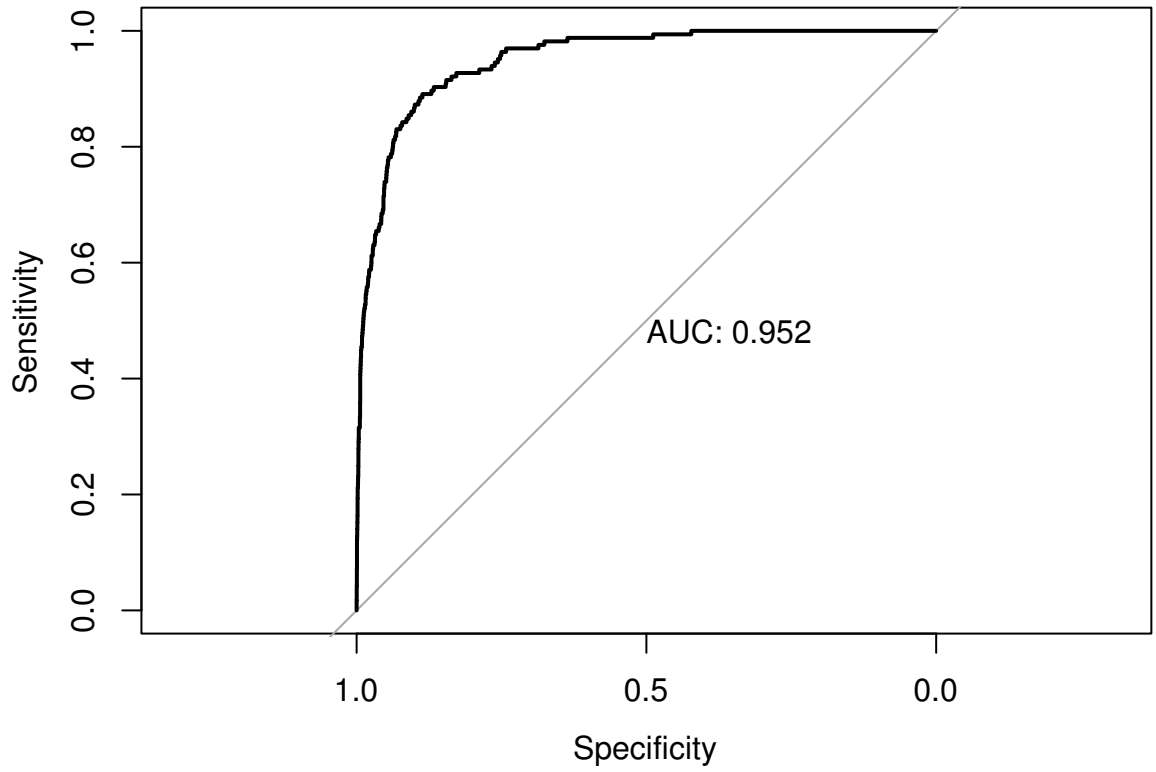
```
## c = 0.50   0.9738  0.31515152   0.9962771
## c = 0.90   0.9674  0.01818182   0.9997932
```

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_tst, type = "response")
test_roc = roc(default_tst$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.

$$P(Y = k \mid X = x) = \frac{e^{\beta_{0k} + \beta_{1k} x_1 + \cdots + + \beta_{pk} x_p}}{\sum_{g=1}^{G} e^{\beta_{0g} + \beta_{1g} x_1 + \cdots + \beta_{pg} x_p}}$$

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_idx = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_trn = iris[iris_idx, ]
iris_test = iris[-iris_idx, ]
```

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_trn, 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_trn))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(model_multi, newdata = iris_trn, type = "prob"))
```

```
##           setosa   versicolor    virginica
## 23  1.000000e+00 2.607782e-19 3.891079e-44
## 106 4.461651e-38 2.328295e-09 1.000000e+00
## 37  1.000000e+00 1.108222e-18 1.620112e-42
## 40  1.000000e+00 5.389221e-15 1.525649e-37
## 145 1.146554e-28 9.816687e-07 9.999990e-01
## 36  1.000000e+00 6.216549e-16 7.345269e-40
```

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.

10.6 rmarkdown

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

```
## [1] "nnet"    "pROC"    "caret"   "ggplot2" "lattice" "tibble"  "ISLR"
```

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.

$$p_k(x) = P(Y = k \mid X = x) = \frac{\pi_k \cdot f_k(x)}{\sum_{g=1}^{G} \pi_g \cdot f_g(x)}$$

We call $p_k(x)$ the **posterior** probability, which we will estimate then use to create classifications. The π_g are called the **prior** probabilities for each possible class g. That is, $\pi_g = P(Y = g)$, unconditioned on X. The $f_g(x)$ are called the **likelihoods**, which are indexed by g 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, $f_g(x)$. 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

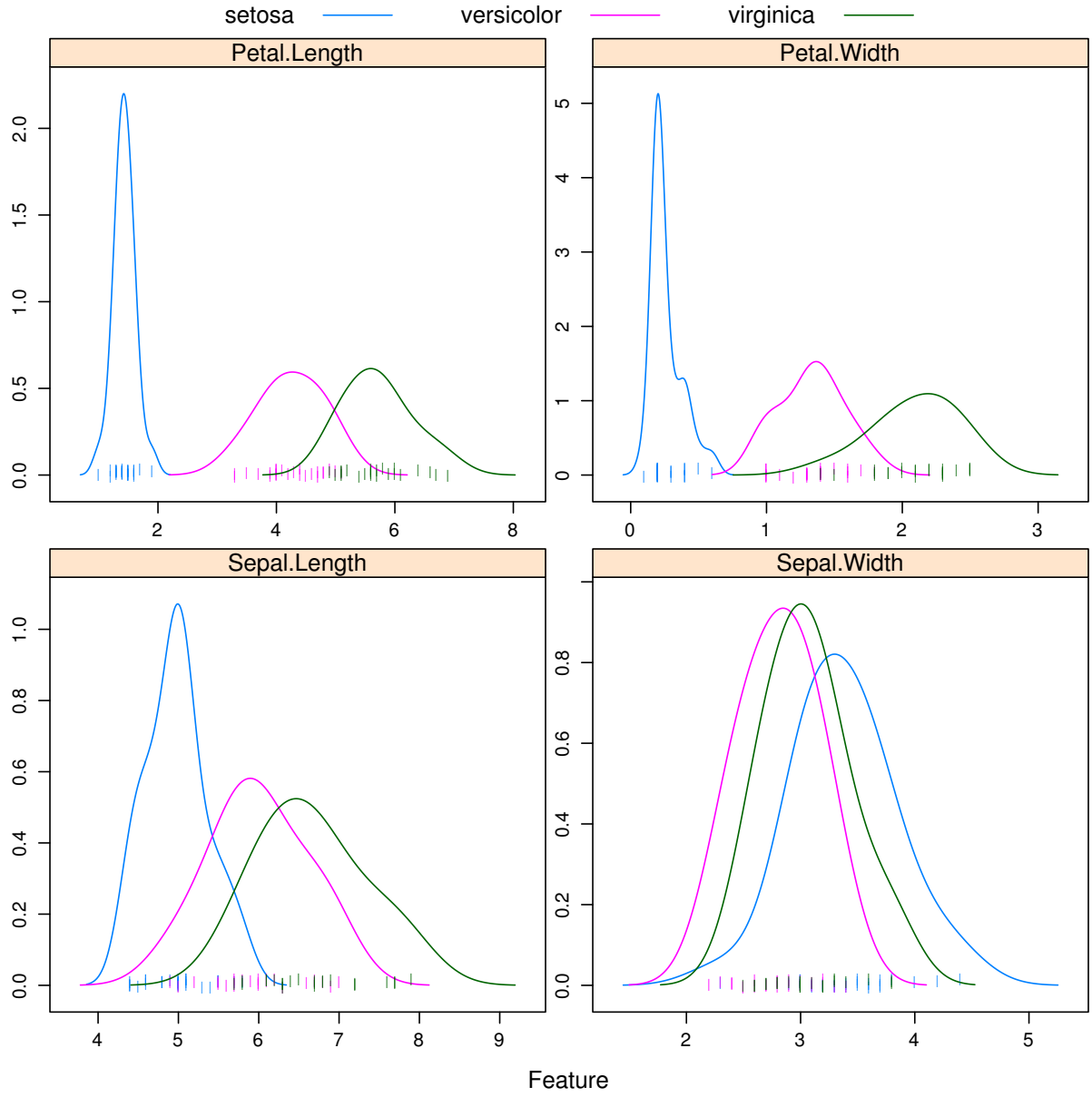
$$\log(\hat{\pi}_k \cdot \hat{f}_k(\mathbf{x})).$$

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.

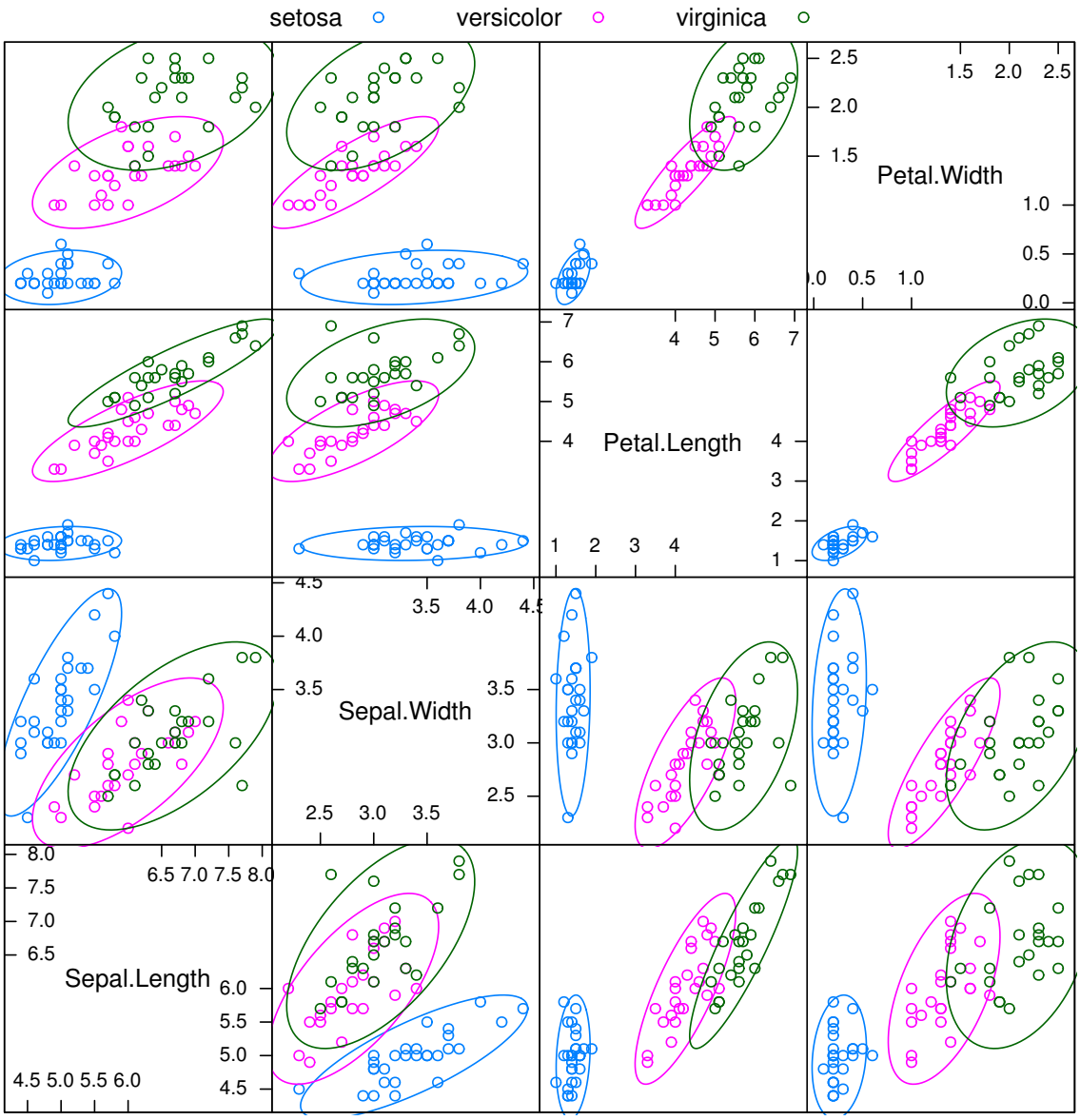
```
set.seed(430)
iris_obs = nrow(iris)
iris_idx = sample(iris_obs, size = trunc(0.50 * iris_obs))
# iris_index = sample(iris_obs, size = trunc(0.10 * iris_obs))
iris_trn = iris[iris_idx, ]
iris_tst = iris[-iris_idx, ]
```

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



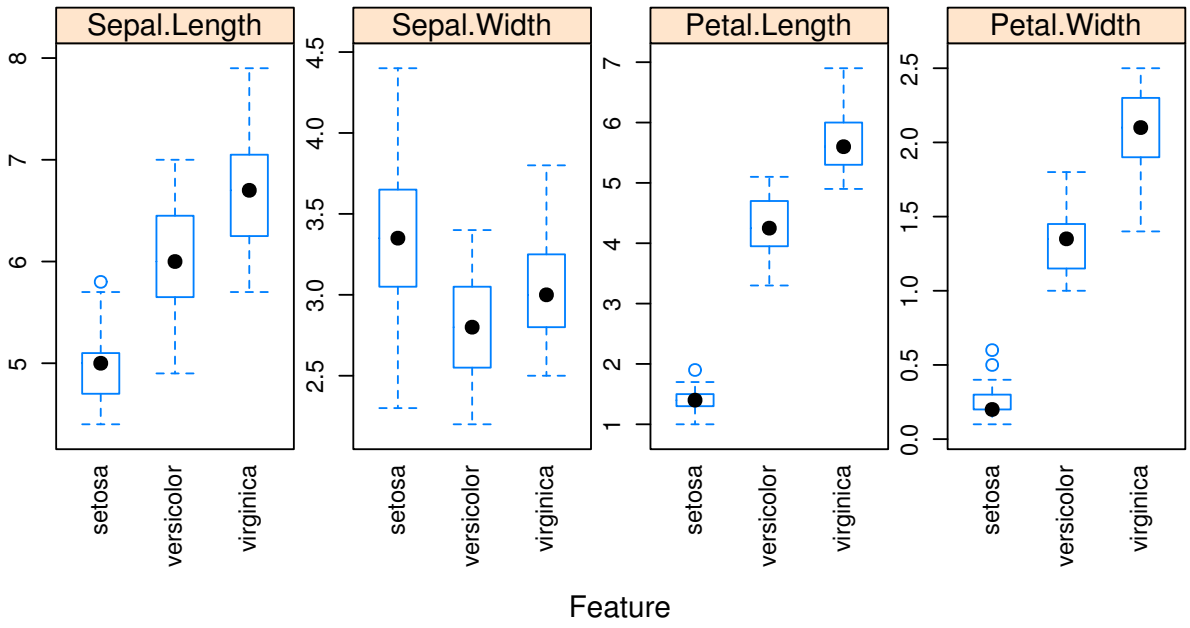

```
caret::featurePlot(x = iris_trn[, c("Sepal.Length", "Sepal.Width",
                                    "Petal.Length", "Petal.Width")],
                   y = iris_trn$Species,
```

```
                    plot = "ellipse",
                    auto.key = list(columns = 3))
```



Scatter Plot Matrix

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

$$X \mid Y = k \sim N(\mu_k, \Sigma)$$

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_k)'\Sigma^{-1}(\mathbf{x}-\mu_k)\right]$$

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_trn)
iris_lda
```

```
## Call:
## lda(Species ~ ., data = iris_trn)
##
## 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_trn))
```

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

```
names(predict(iris_lda, iris_trn))
```

```
## [1] "class"     "posterior" "x"
```

```
head(predict(iris_lda, iris_trn)$class, n = 10)
```

```
##  [1] setosa     virginica  setosa     setosa     virginica  setosa
##  [7] virginica  setosa     versicolor setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_lda, iris_trn)$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_trn_pred = predict(iris_lda, iris_trn)$class
iris_lda_tst_pred = predict(iris_lda, iris_tst)$class
```

We store the predictions made on the train and test sets.

```
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

```
calc_class_err(predicted = iris_lda_trn_pred, actual = iris_trn$Species)
```

```
## [1] 0.04
```

```
calc_class_err(predicted = iris_lda_tst_pred, actual = iris_tst$Species)
```

```
## [1] 0.01333333
```

As expected, LDA performs well on both the train and test data.

```
table(predicted = iris_lda_tst_pred, actual = iris_tst$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_trn, prior = c(1, 1, 1) / 3)
iris_lda_flat
```

```
## Call:
## lda(Species ~ ., data = iris_trn, 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_trn_pred = predict(iris_lda_flat, iris_trn)$class
iris_lda_flat_tst_pred = predict(iris_lda_flat, iris_tst)$class
```

```
calc_class_err(predicted = iris_lda_flat_trn_pred, actual = iris_trn$Species)
```

```
## [1] 0.04
```

```
calc_class_err(predicted = iris_lda_flat_tst_pred, actual = iris_tst$Species)
```

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

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.

$$X \mid Y = k \sim N(\mu_k, \Sigma_k)$$

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu_k)'\Sigma_k^{-1}(\mathbf{x}-\mu_k)\right]$$

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_trn)
iris_qda
```

```
## Call:
## qda(Species ~ ., data = iris_trn)
##
## 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_trn_pred = predict(iris_qda, iris_trn)$class
iris_qda_tst_pred = predict(iris_qda, iris_tst)$class
```

The `predict()` function operates the same as the `predict()` function for LDA.

```
calc_class_err(predicted = iris_qda_trn_pred, actual = iris_trn$Species)
```

```
## [1] 0.01333333
```

```
calc_class_err(predicted = iris_qda_tst_pred, actual = iris_tst$Species)
```

```
## [1] 0.04
```

```
table(predicted = iris_qda_tst_pred, actual = iris_tst$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.

$$f_k(x) = \prod_{j=1}^{j=p} f_{kj}(x_j)$$

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_trn)
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_trn))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_nb, iris_trn, type = "class"))
```

```
## [1] setosa    virginica setosa    setosa    virginica setosa
## Levels: setosa versicolor virginica
```

```
head(predict(iris_nb, iris_trn, 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_trn_pred = predict(iris_nb, iris_trn)
iris_nb_tst_pred = predict(iris_nb, iris_tst)
```

```
calc_class_err(predicted = iris_nb_trn_pred, actual = iris_trn$Species)
```

```
## [1] 0.05333333
```

```
calc_class_err(predicted = iris_nb_tst_pred, actual = iris_tst$Species)
```

```
## [1] 0.05333333
```

```
table(predicted = iris_nb_tst_pred, actual = iris_tst$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 Error	Test Error
LDA	0.0400000	0.0133333
LDA, Flat Prior	0.0400000	0.0000000
QDA	0.0133333	0.0400000
Naive Bayes	0.0533333	0.0533333

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_trn_mod = iris_trn

iris_trn_mod$Sepal.Width = ifelse(iris_trn$Sepal.Width > 3,
                                  ifelse(iris_trn$Sepal.Width > 4,
                                         "Large", "Medium"),
                                  "Small")

unique(iris_trn_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_trn_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_trn_mod)
```

```
## Call:
## lda(Species ~ Sepal.Length + Sepal.Width, data = iris_trn_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 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

Chapter 12

k-Nearest Neighbors

In this chapter we introduce our first **non-parametric** classification method, k-nearest neighbors. So far, all of the methods for classificaiton that we have seen have been parametric. For example, logistic regression had the form

$$\log\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

In this case, the β_j are the parameters of the model, which we learned (estimated) by training (fitting) the model. Those estimates were then used to obtain estimates of the probability $p(x) = P(Y = 1 \mid X = x)$,

$$\hat{p}(x) = \frac{e^{\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p}}{1 + e^{\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p}}$$

As we saw in regression, k-nearest neighbors has no such model 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 with non-parametric methods. Later we will see examples of tuning parameters for parametric methods.

Often when discussing k-nearest neighbors for classification, it is framed as a black-box method that directly returns classifications. We will instead frame it as a non-parametric model for the probabilites $p_g(x) = P(Y = g \mid X = x)$. That is a k-nearest neighbors model using k neighbors estimates this probability as

$$\hat{p}_{kg}(x) = \hat{P}_k(Y = g \mid X = x) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(x, \mathcal{D})} I(y_i = g)$$

Essentially, the probability of each class g is the proportion of the k neighbors of x with that class, g.

Then, to create a classifier, as always, we simply classify to the class with the highest estimated probability.

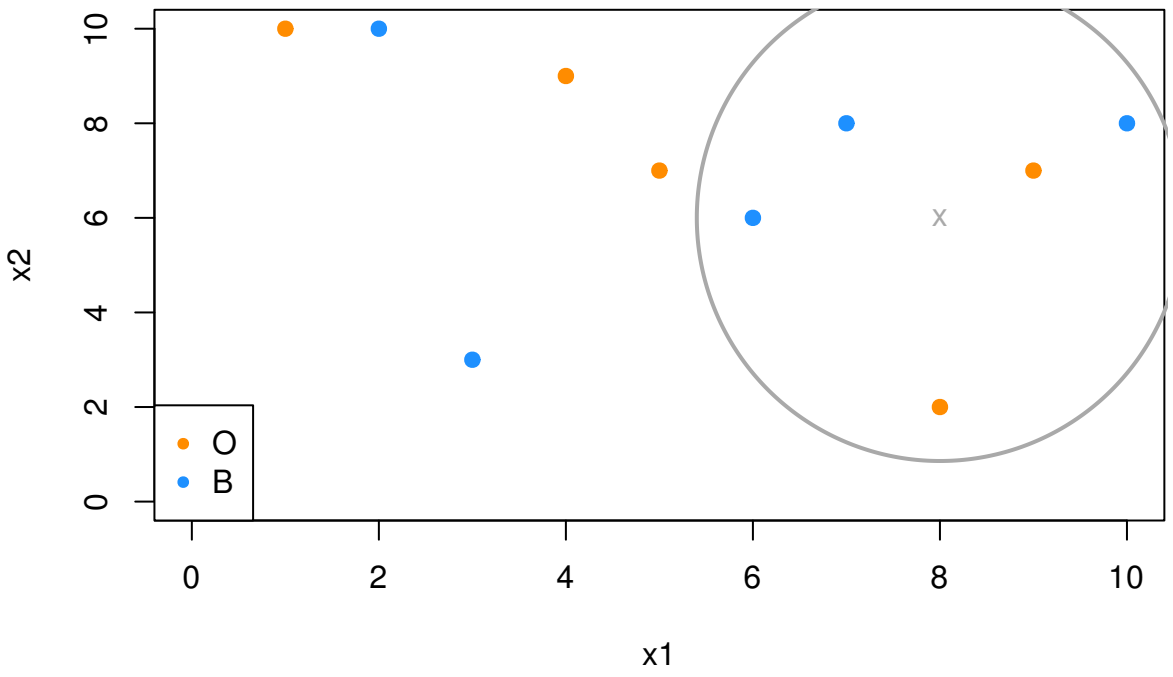
$$\hat{C}_k(x) = \underset{g}{\text{argmax}} \ \ \hat{p}_{kg}(x)$$

This is the same as saying that we classify to the class with the most observations in the k nearest neighbors. If more than one class is tied for the highest estimated probablity, simply assign a class at random to one of the classes tied for highest.

In the binary case this becomes

$$\hat{C}_k(x) = \begin{cases} 1 & \hat{p}_{k0}(x) > 0.5 \\ 0 & \hat{p}_{k1}(x) < 0.5 \end{cases}$$

Again, if the probability for class `0` and `1` are equal, simply assign at random.



In the above example, when predicting at $x = (x_1, x_2) = (8, 6)$,

$$\hat{p}_{5B}(x_1 = 8, x_2 = 6) = \hat{P}_5(Y = \text{Blue} \mid X_1 = 8, X_2 = 6) = \frac{3}{5}$$

$$\hat{p}_{5O}(x_1 = 8, x_2 = 6) = \hat{P}_5(Y = \text{Orange} \mid X_1 = 8, X_2 = 6) = \frac{2}{5}$$

Thus

$$\hat{C}_5(x_1 = 8, x_2 = 6) = \text{Blue}$$

12.1 Binary Data Example

```
library(ISLR)
library(class)
```

We first load some necessary libraries. We'll begin discussing k-nearest neighbors for classification by returning to the `Default` data from the `ISLR` package. To perform k-nearest neighbors for classification, we will use the `knn()` function from the `class` package.

Unlike many of our previous methods, such as logistic regression, `knn()` requires that all predictors be numeric, so we coerce `student` to be a `0` and `1` dummy variable instead of a factor. (We can, and should, leave the response as a factor.) Numeric predictors are required because of the distance calculations taking place.

```
set.seed(42)
Default$student = as.numeric(Default$student) - 1
default_idx = sample(nrow(Default), 5000)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

Like we saw with `knn.reg` form the `FNN` package for regression, `knn()` from `class` 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.

Note that the `FNN` package also contains a `knn()` function for classification. We choose `knn()` from `class` as it seems to be much more popular. However, you should be aware of which packages you have loaded and thus which functions you are using. They are very similar, but have some differences.

```
# training data
X_default_trn = default_trn[, -1]
y_default_trn = default_trn$default

# testing data
X_default_tst = default_tst[, -1]
y_default_tst = default_tst$default
```

Again, 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 observation, the method must find the k-nearest neighbors, which is not computationally cheap. Note that by deafult, `knn()` uses Euclidean distance to determine neighbors.

```
head(knn(train = X_default_trn,
         test  = X_default_tst,
         cl    = y_default_trn,
         k     = 3))
```

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

Because of the lack of any need for training, the `knn()` function immediately returns classifications. With logistic regression, we needed to use `glm()` to fit the model, then `predict()` to obtain probabilities we would use to make a classifier. Here, the `knn()` function directly returns classifications. That is `knn()` is essentially $\hat{C}_k(x)$.

Here, `knn()` takes four arguments:

- `train`, the predictors for the train set.
- `test`, the predictors for the test set. `knn()` will output results (classifications) for these cases.
- `cl`, the true class labels for the train set.
- `k`, the number of neighbors to consider.

```
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
```

```
}
```

We'll use our usual `calc_class_err()` function to asses how well `knn()` works with this data. We use the test data to evaluate.

```
calc_class_err(actual    = y_default_tst,
               predicted = knn(train = X_default_trn,
                               test  = X_default_tst,
                               cl    = y_default_trn,
                               k     = 5))
```

```
## [1] 0.0316
```

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 a mean of zero and unit variance. Be sure to apply the scaling to both the train and test data.

```
calc_class_err(actual    = y_default_tst,
               predicted = knn(train = scale(X_default_trn),
                               test  = scale(X_default_tst),
                               cl    = y_default_trn,
                               k     = 5))
```

```
## [1] 0.0278
```

Here we see the scaling slightly 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
err_k = rep(x = 0, times = length(k_to_try))

for (i in seq_along(k_to_try)) {
  pred = knn(train = scale(X_default_trn),
             test  = scale(X_default_tst),
             cl    = y_default_trn,
             k     = k_to_try[i])
  err_k[i] = calc_class_err(y_default_tst, pred)
}
```

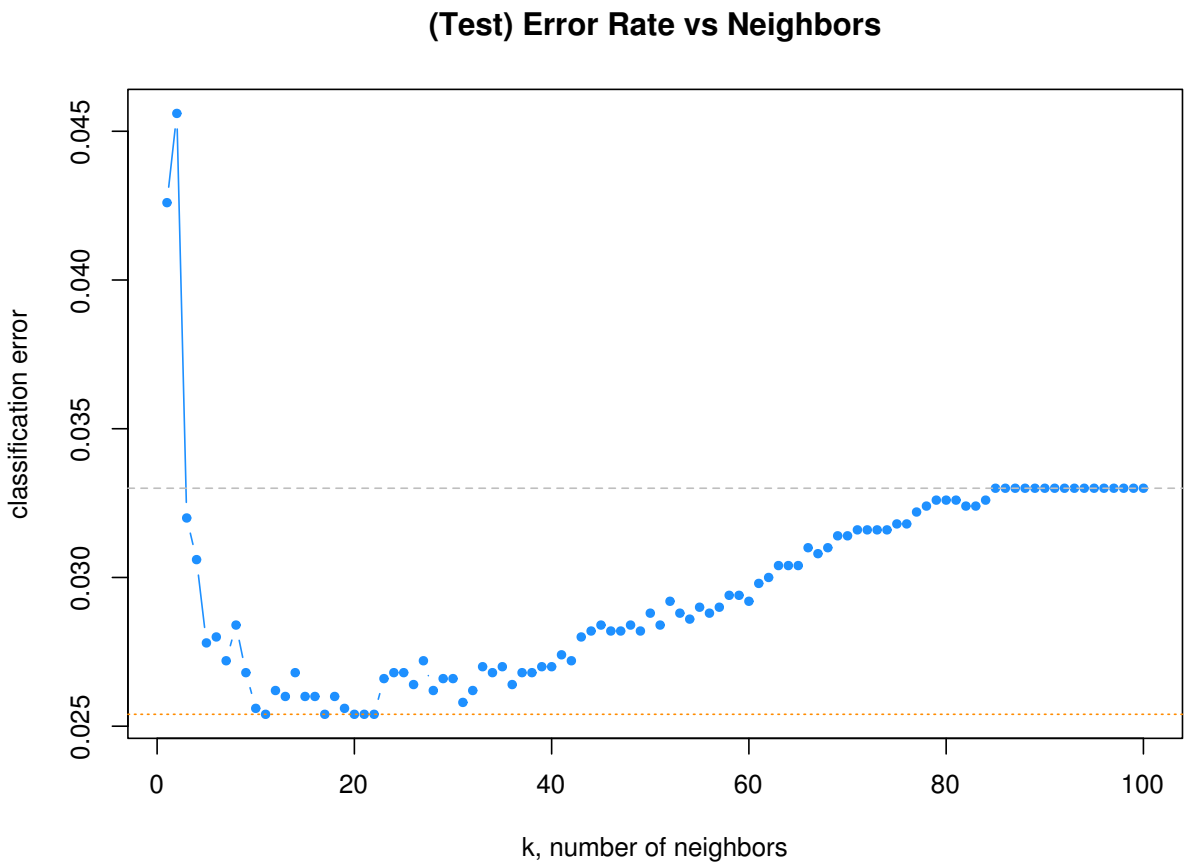
The `seq_along()` function can be very useful for looping over a vector that stores non-consecutive numbers. It often removes the need for an additional counter variable. We actually didn't need it in the above `knn()` example, but it is still a good habit. For example maybe we didn't want to try every value of k, but only odd integers, which woudl prevent ties. Or perhaps we'd only like to check multiples of 5 to further cut down on computation time.

Also, note that we set a seed before running this loops. This is because we are considering even values of k, thus, there are ties which are randomly broken.

Naturally, we plot the k-nearest neighbor results.

```
# plot error vs choice of k
plot(err_k, type = "b", col = "dodgerblue", cex = 1, pch = 20,
     xlab = "k, number of neighbors", ylab = "classification error",
     main = "(Test) Error Rate vs Neighbors")
```

```
# add line for min error seen
abline(h = min(err_k), col = "darkorange", lty = 3)
# add line for minority prevalence in test set
abline(h = mean(y_default_tst == "Yes"), col = "grey", lty = 2)
```



The dotted orange line represents the smallest observed test classification error rate.

```
min(err_k)
```

```
## [1] 0.0254
```

We see that five different values of k are tied for the lowest error rate.

```
which(err_k == min(err_k))
```

```
## [1] 11 17 20 21 22
```

Given a choice of these five values of k, we select the largest, as it is the least variable, and has the least chance of overfitting.

```
max(which(err_k == min(err_k)))
```

```
## [1] 22
```

Recall that defaulters are the minority class. That is, the majority of observations are non-defaulters.

```
table(y_default_tst)
```

```
## y_default_tst
##   No  Yes
## 4835  165
```

Notice that, as k increases, eventually the error approaches the test prevalence of the minority class.

```
mean(y_default_tst == "Yes")
```

```
## [1] 0.033
```

12.2 Categorical Data

Like LDA and QDA, KNN can be used for both binary and multi-class problems. As an example of a multi-class problems, we return to the `iris` data.

```
set.seed(430)
iris_obs = nrow(iris)
iris_idx = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_trn = iris[iris_idx, ]
iris_tst = iris[-iris_idx, ]
```

All the predictors here are numeric, so we proceed to splitting the data into predictors and classes.

```
# training data
X_iris_trn = iris_trn[, -5]
y_iris_trn = iris_trn$Species

# testing data
X_iris_tst = iris_tst[, -5]
y_iris_tst = iris_tst$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_trn),
                test  = scale(X_iris_tst),
                cl    = y_iris_trn,
                k     = 10,
                prob  = TRUE)
```

```
head(iris_pred, n = 50)
```

```
##  [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
## Levels: setosa versicolor virginica
```

Unfortunately, this only returns the predicted probability of the most common class. In the binary case, this would be sufficient to recover all probabilities, however, for multi-class problems, we cannot recover each of the probabilities of interest. This will simply be a minor annoyance for now, which we will fix when we introduce the `caret` package for model training.

```
head(attributes(iris_pred)$prob, n = 50)
```

```
##  [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
```

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 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

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:
##                           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
```

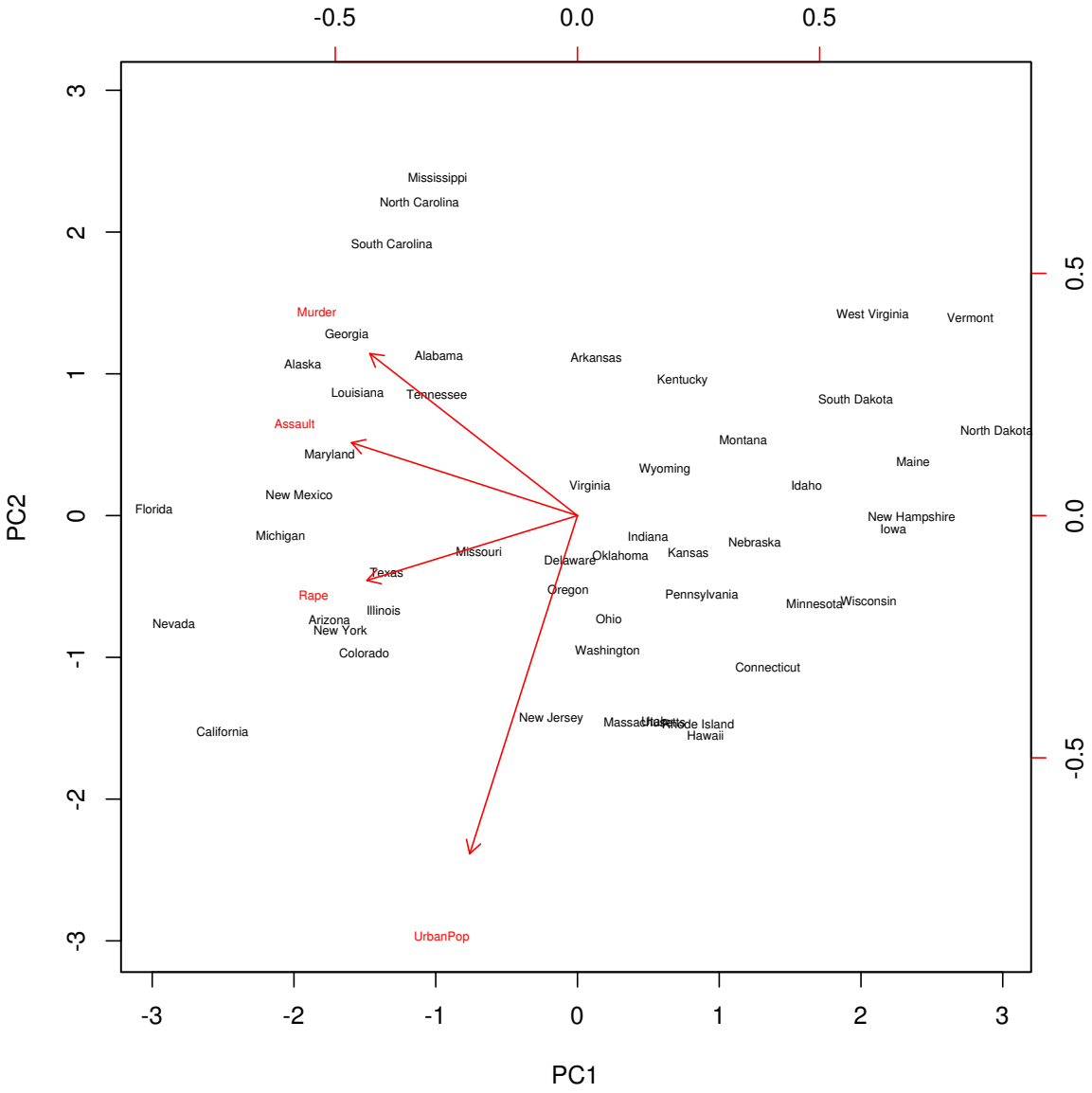
```
USArrests_pca$x[, 1] %*% USArrests_pca$x[, 3]
```

```
##               [,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.

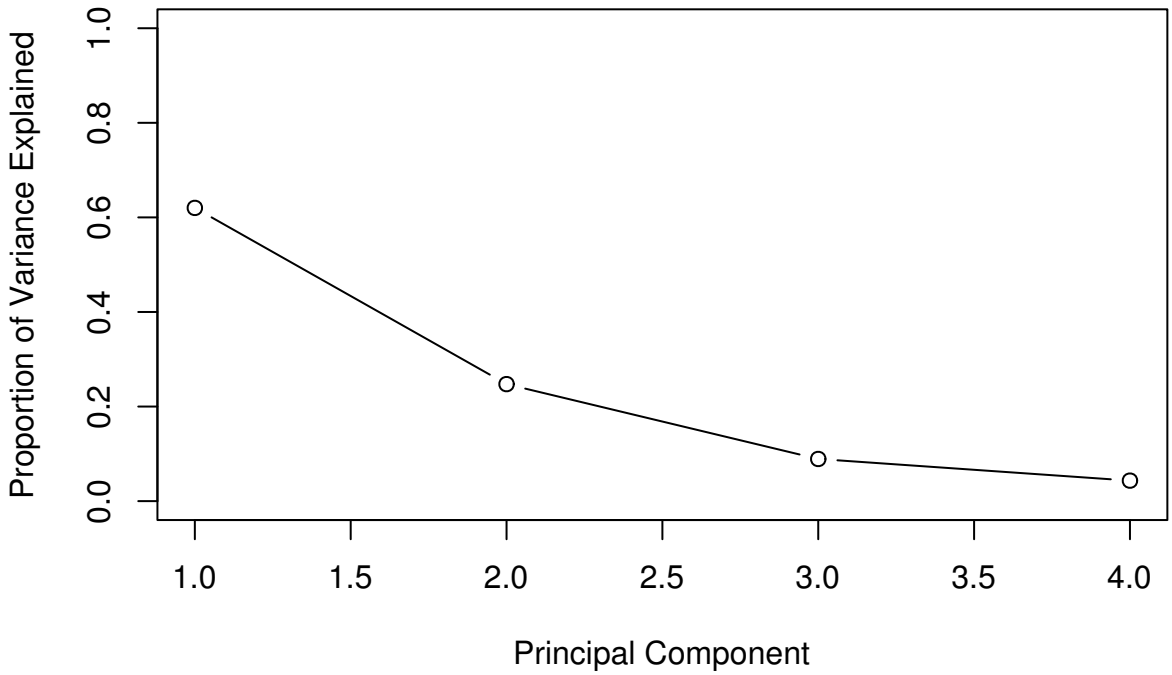
```
get_PVE = function(pca_out) {
  pca_out$sdev ^ 2 / sum(pca_out$sdev ^ 2)
}
```

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'
)
```



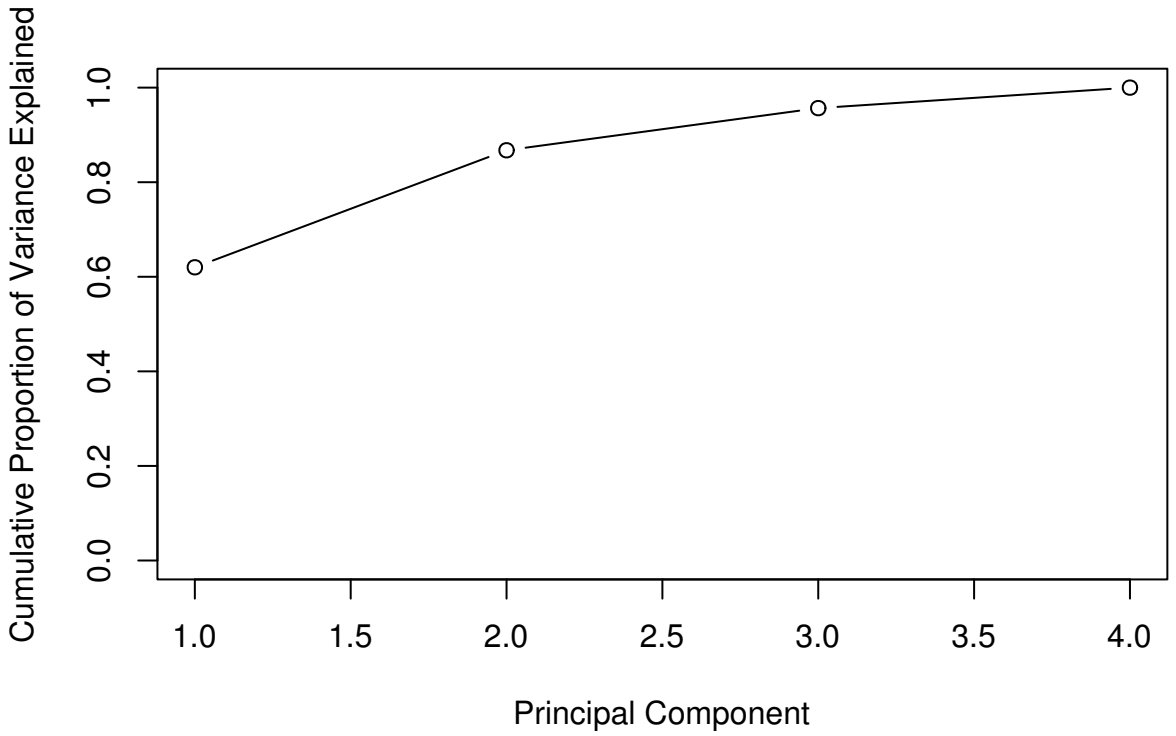
We can then plot the proportion of variance explained for each PC. As expected, we see the PVE decrease.

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

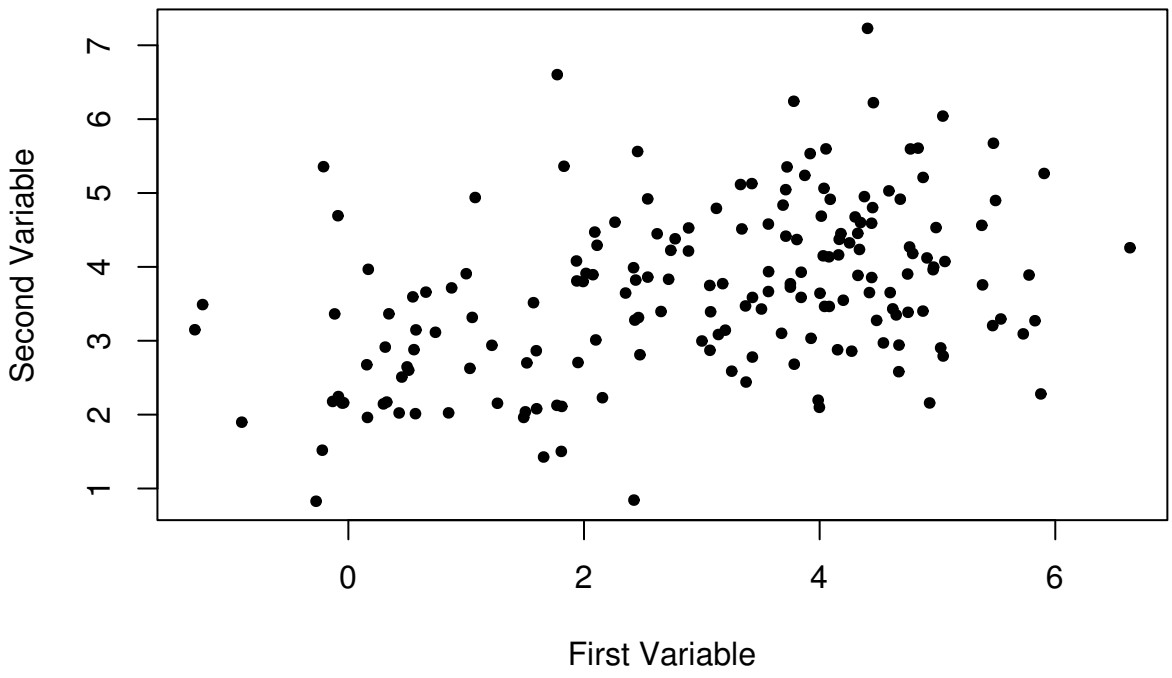
We check how well the clustering is working.

```
dim(clust_data)
```

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

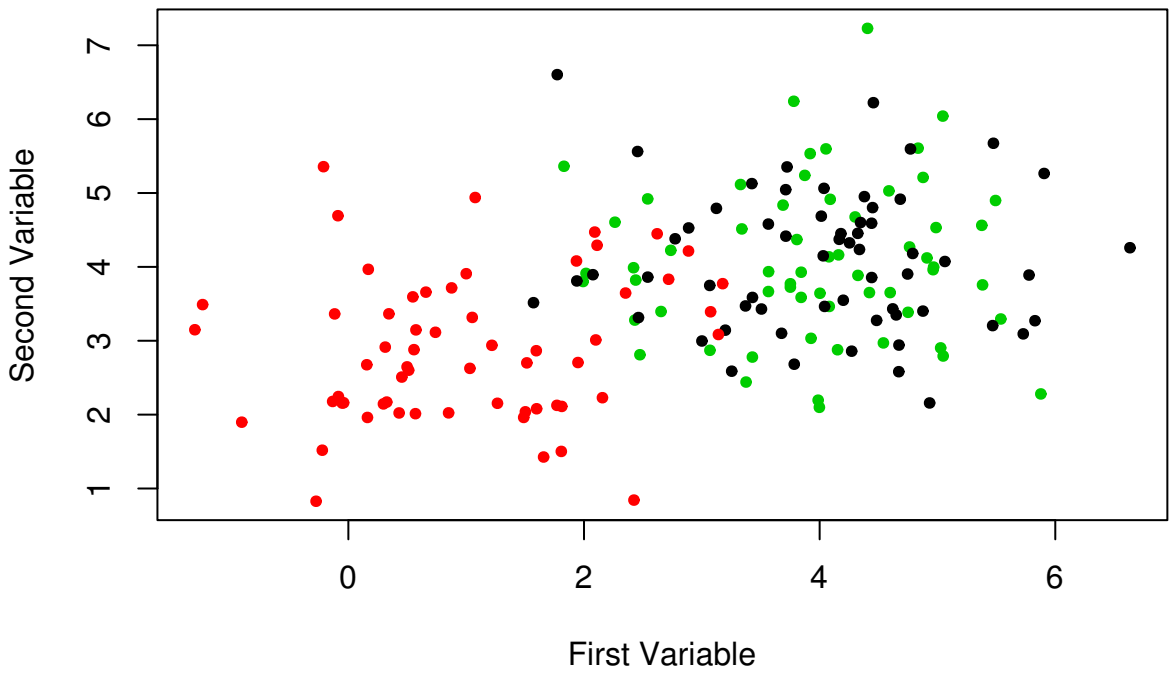
This data is "high dimensional" so it is difficult to visualize. (Anything more than 2 is hard to visualize.)

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



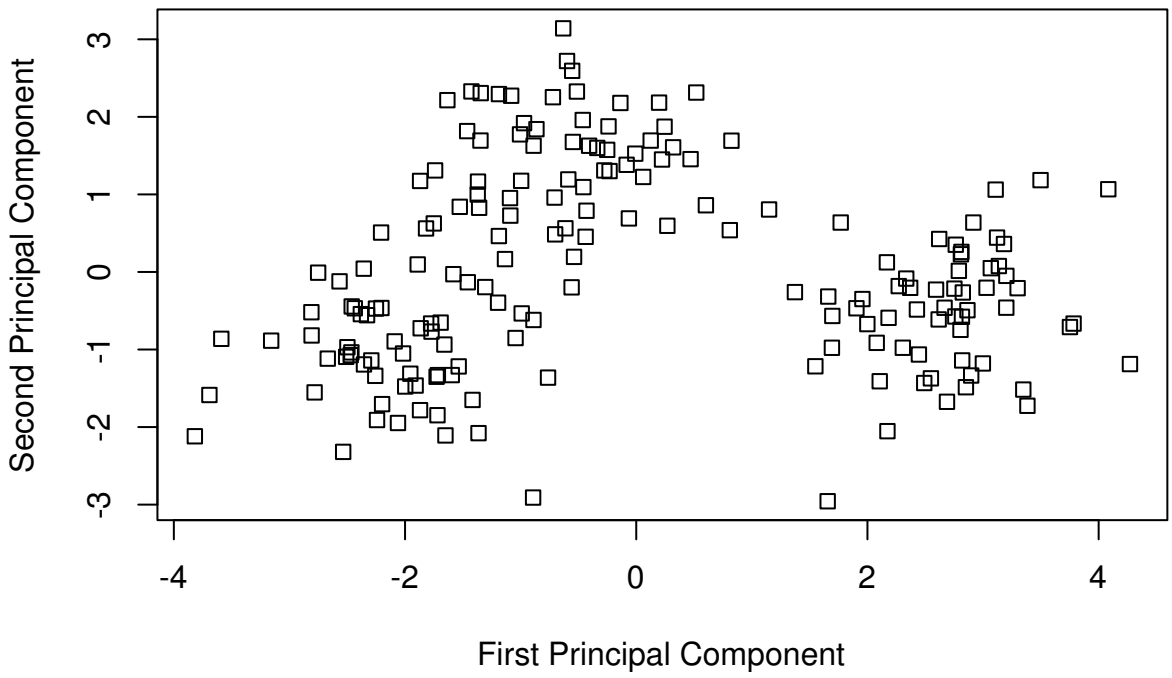
Plotting the first and second variables simply results in a blob.

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



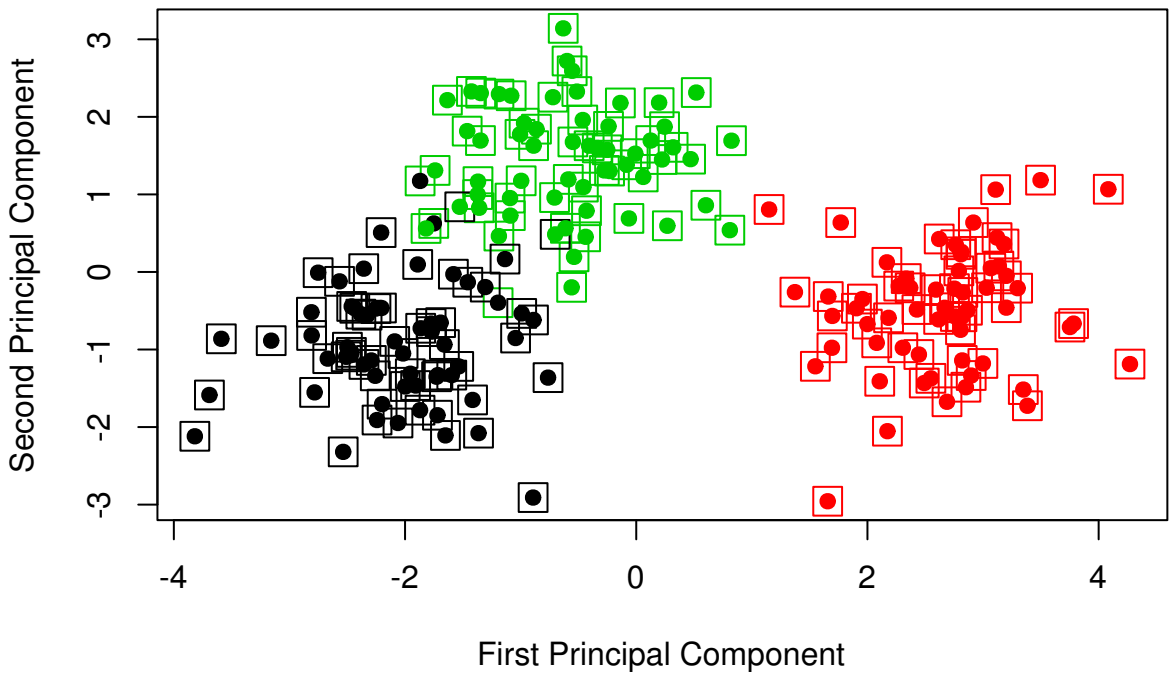
Even when using their true clusters for coloring, this plot isn't very helpful.

```
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"
)
```



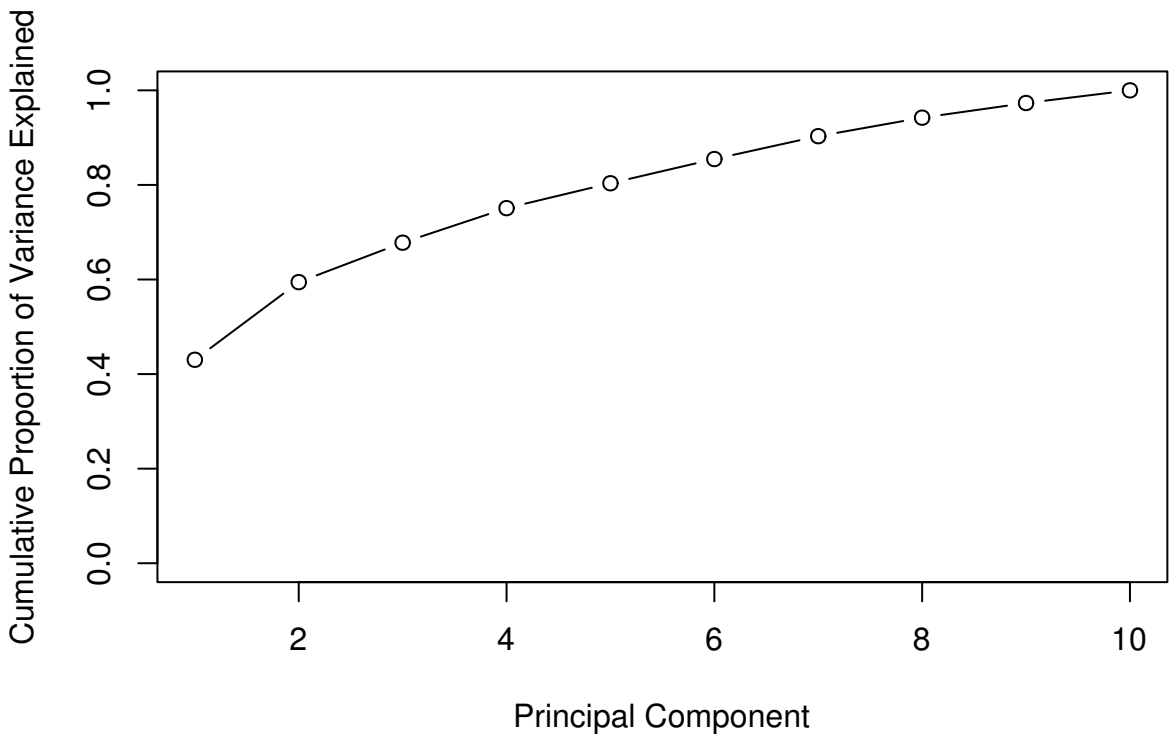
If we instead plot the first two principal components, we see, even without coloring, one blob that is clearly separate from the rest.

```
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'
)
```

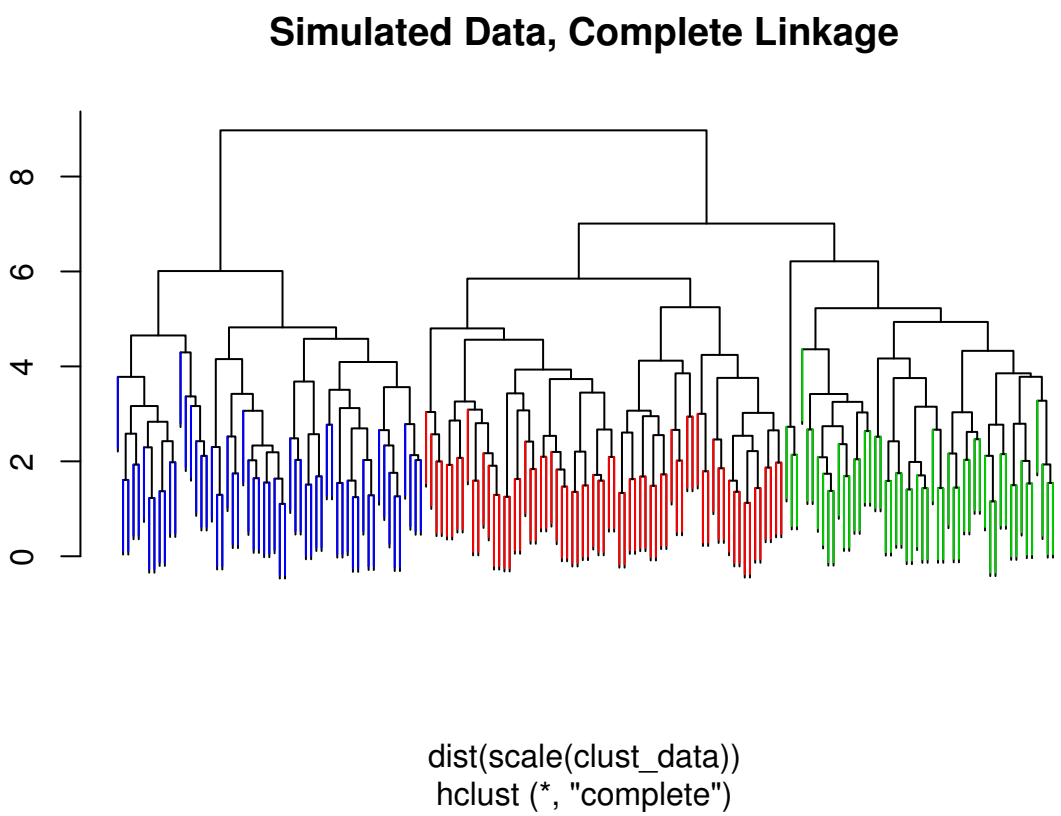


The above visualization works well because the first two PCs explain a large proportion of the variance.

```
#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)
```



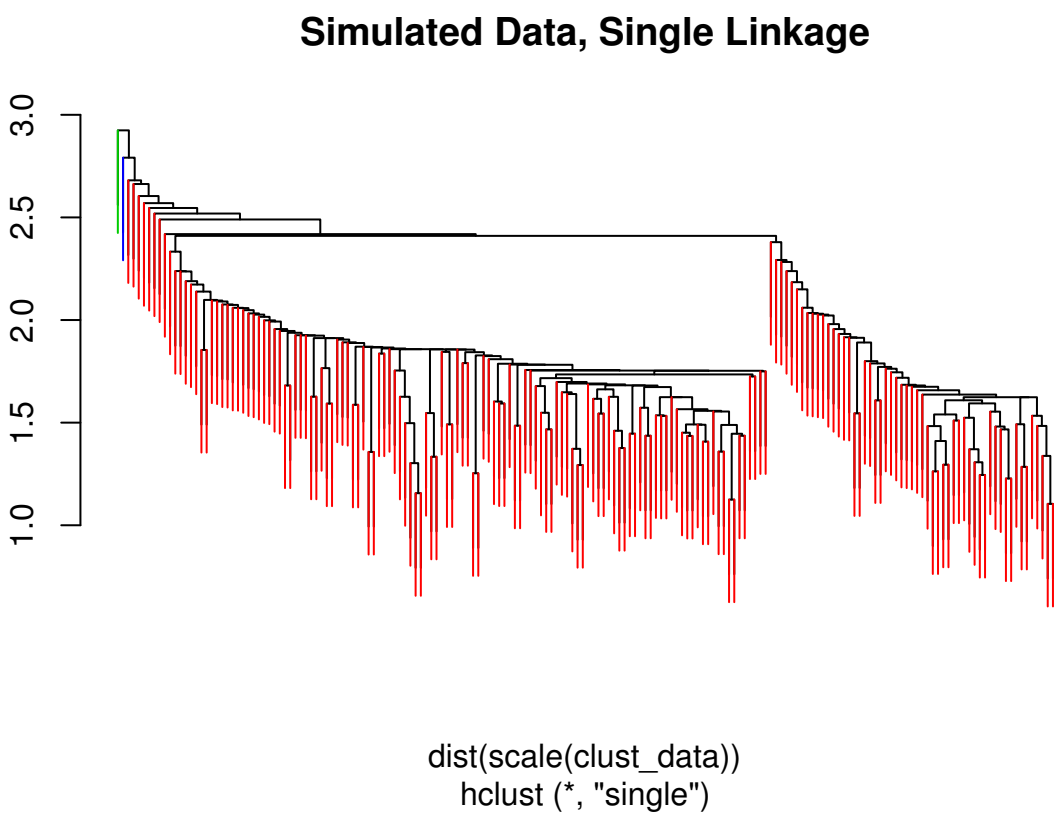
Here we apply hierarchical clustering to the **scaled** data. The `dist()` function is used to calculate pairwise distances between the (scaled in this case) observations. We use complete linkage. We then use the `cutree()` function to cluster the data into `3` clusters. The `ColorDendrogram()` function 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.

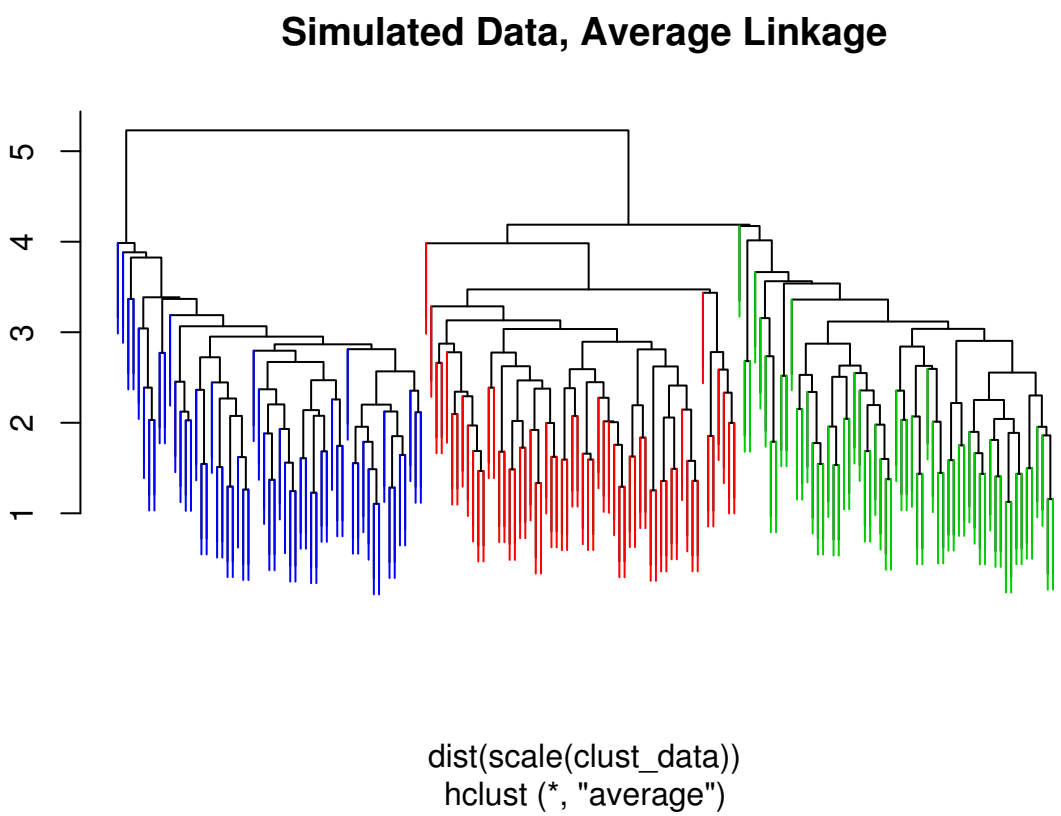
```
clust_data_hc = hclust(dist(scale(clust_data)), method = "single")
clust_data_cut = cutree(clust_data_hc , 3)
ColorDendrogram(clust_data_hc, y = clust_data_cut,
                labels = names(clust_data_cut),
                main = "Simulated Data, Single Linkage",
                branchlength =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

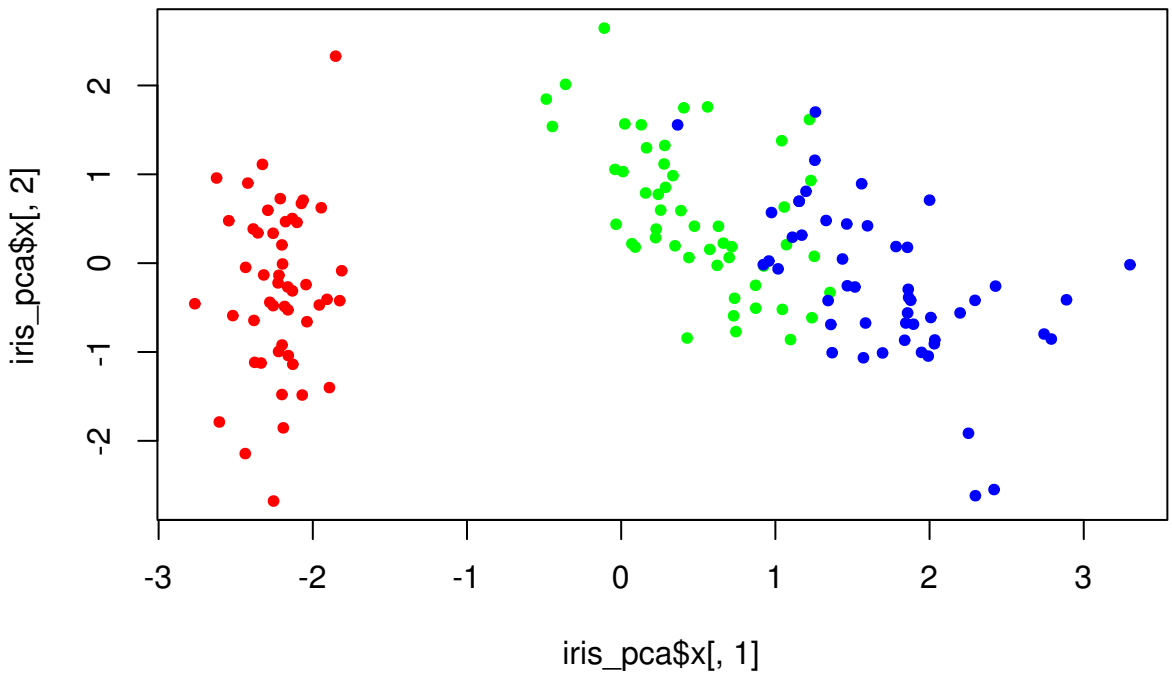
```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 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
```

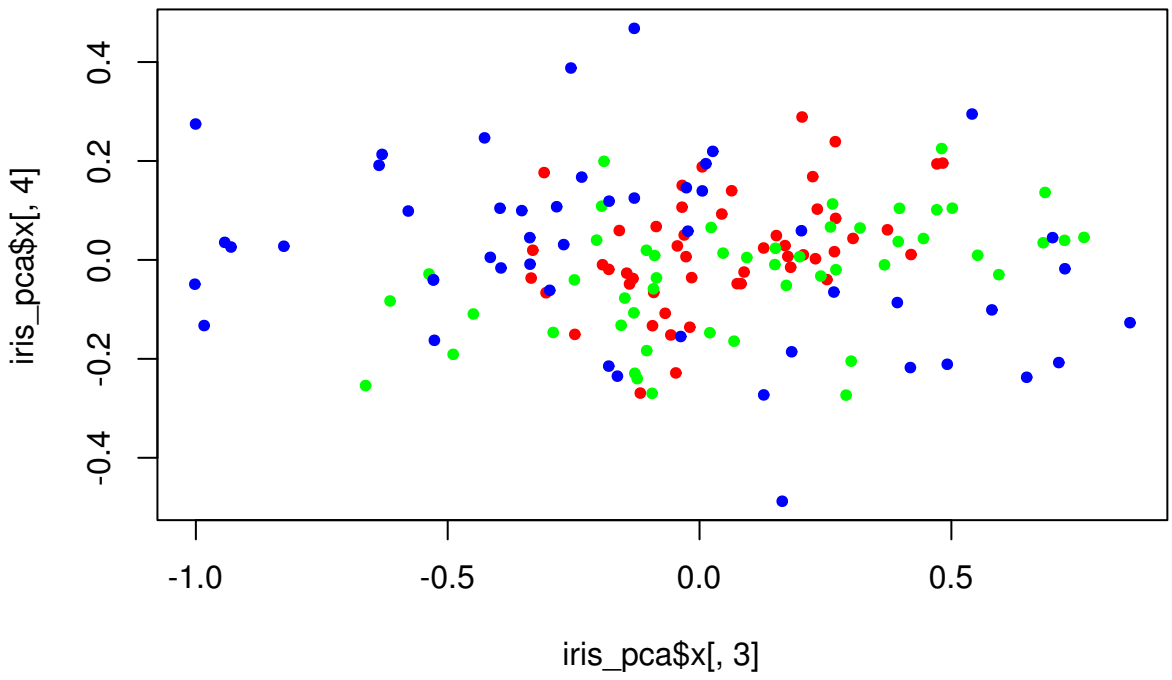
```
##                     PC1         PC2        PC3        PC4
## Sepal.Length  0.5210659 -0.37741762  0.7195664  0.2612863
```

```
## 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
```

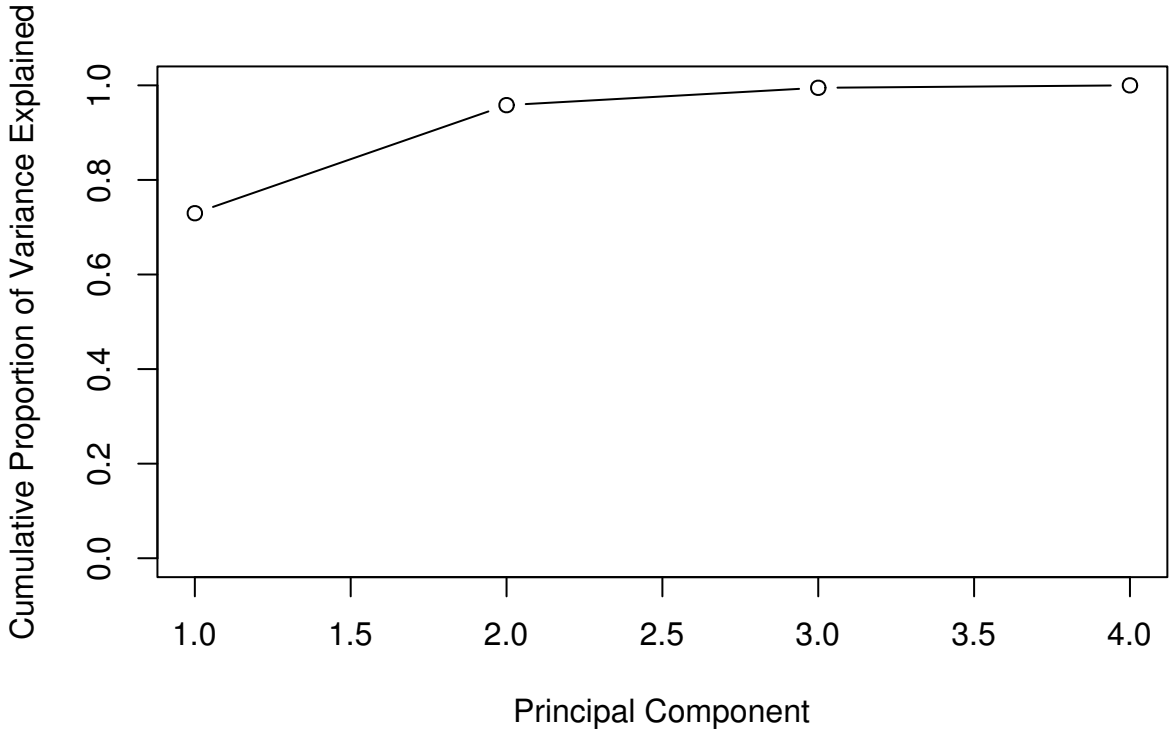
```
lab_to_col = function (labels){
  cols = rainbow (length(unique(labels)))
  cols[as.numeric (as.factor(labels))]
}

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)
```



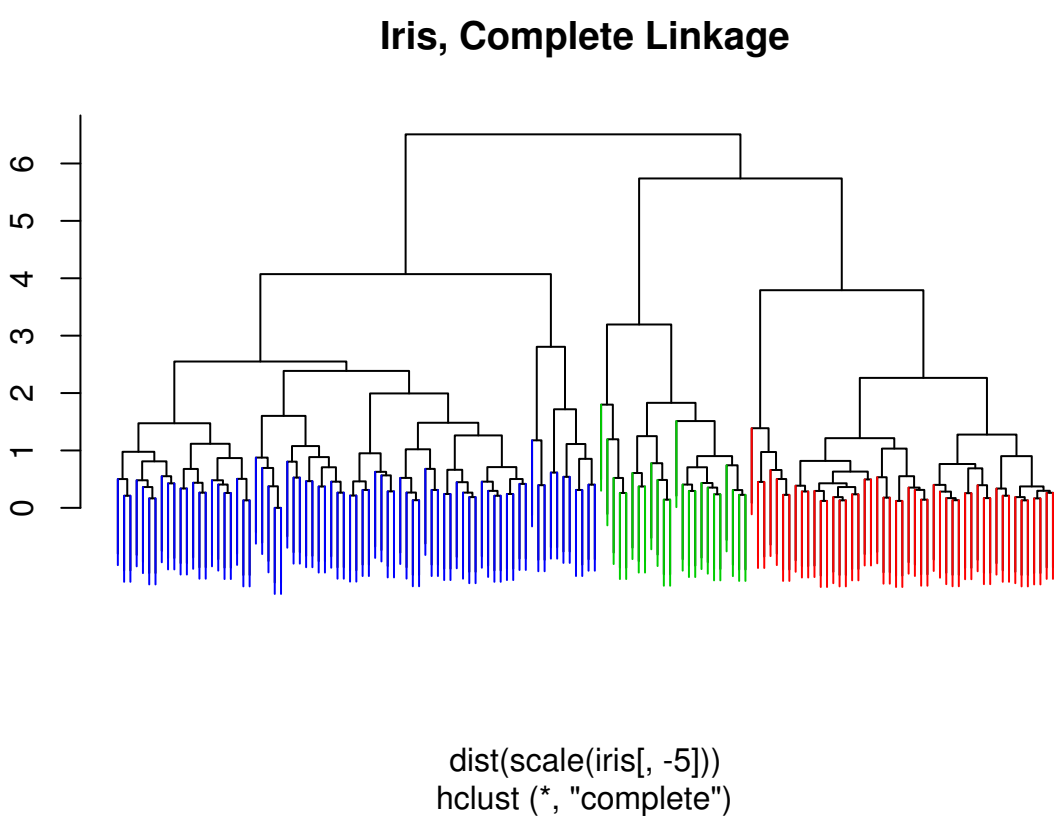

```
iris_pve = get_PVE(iris_pca)

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)
```



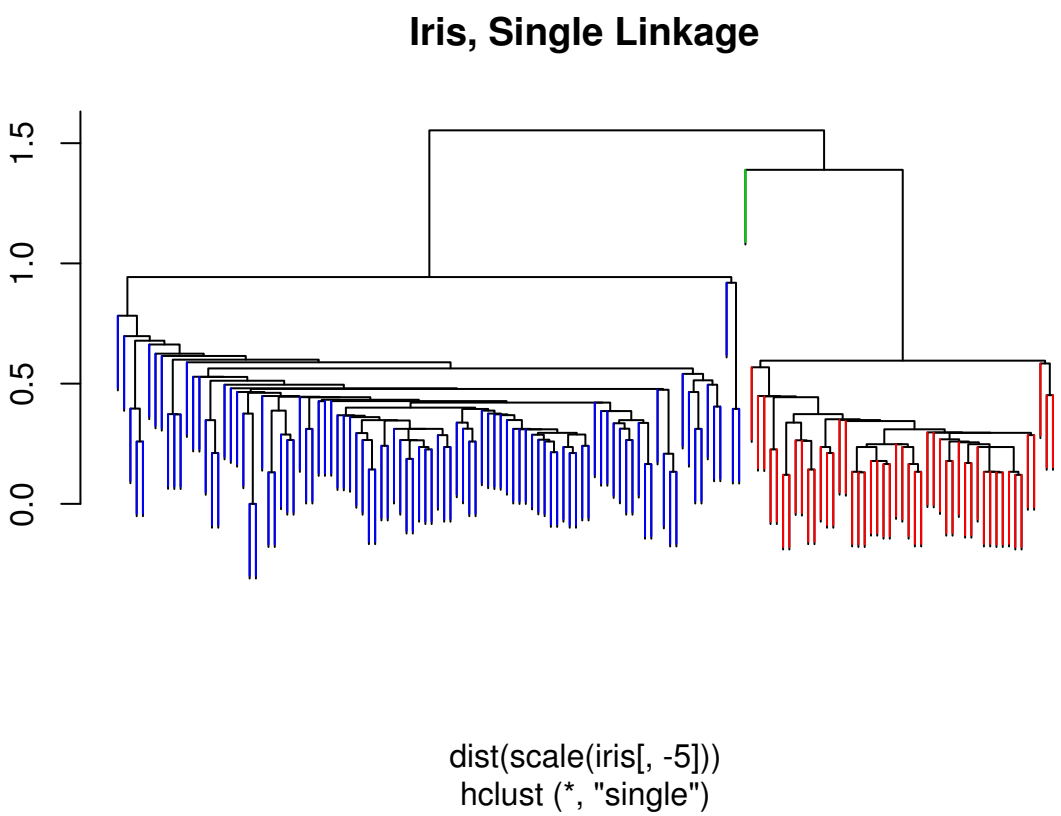

```
table(iris_cut, iris[,5])
```

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

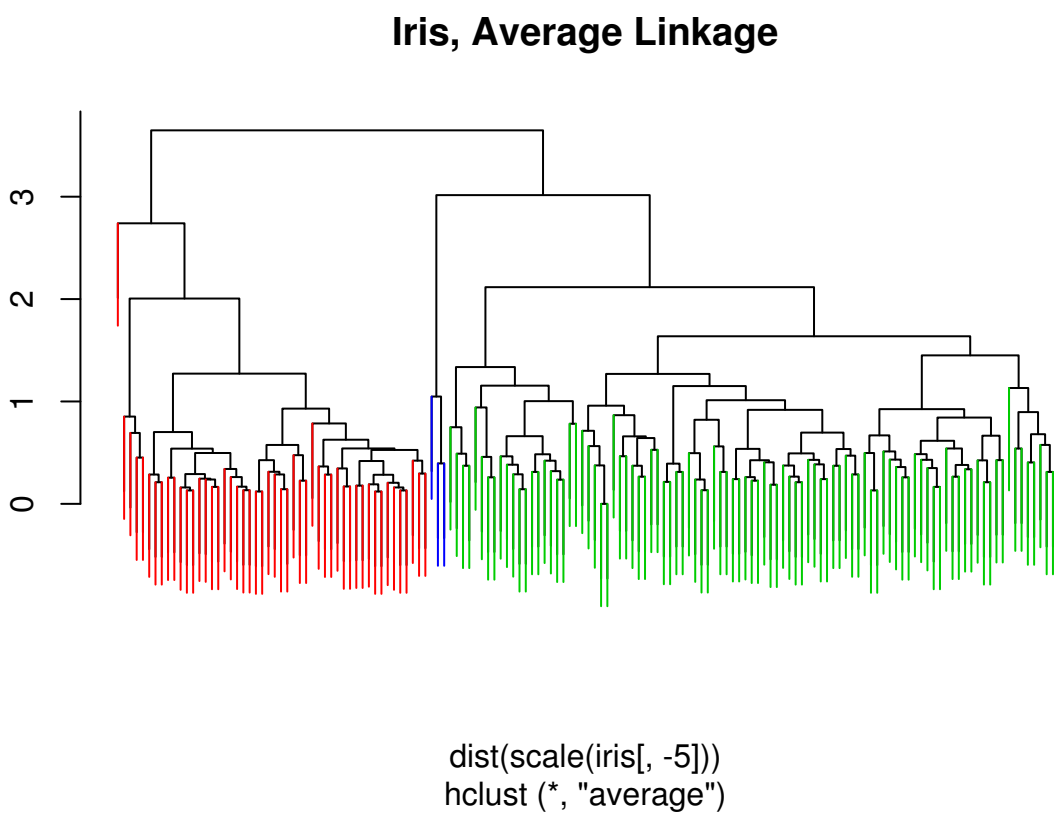
```
table(iris_cut, iris_kmeans$clust)
```

```
##
## 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 3.5.1 and the following packages:

- Base Packages, Attached

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

- Additional Packages, Attached

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

- Additional Packages, Not Attached

```
##  [1] "tidyselect"   "xfun"         "purrr"        "reshape2"
##  [5] "splines"      "colorspace"   "generics"     "stats4"
```

```
##  [9] "htmltools"    "yaml"         "prodlim"      "survival"
## [13] "rlang"        "e1071"        "ModelMetrics" "pillar"
## [17] "glue"         "withr"        "bindrcpp"     "foreach"
## [21] "bindr"        "plyr"         "lava"         "stringr"
## [25] "timeDate"     "munsell"      "gtable"       "recipes"
## [29] "codetools"    "evaluate"     "knitr"        "class"
## [33] "Rcpp"         "scales"       "backports"    "ipred"
## [37] "digest"       "stringi"      "bookdown"     "dplyr"
## [41] "grid"         "rprojroot"    "tools"        "magrittr"
## [45] "lazyeval"     "tibble"       "crayon"       "pkgconfig"
## [49] "Matrix"       "data.table"   "lubridate"    "gower"
## [53] "assertthat"   "rmarkdown"    "iterators"    "R6"
## [57] "rpart"        "nnet"         "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 3.5.1.

Chapter 20

Resampling

- **NOTE**: This chapter is currently be re-written and will likely change considerably in the near future. It is currently lacking in a number of ways mostly narrative.

In this chapter we introduce **resampling** methods, in particular **cross-validation**. We will highlight the need for cross-validation by comparing it to our previous approach, which was to simply set aside some "test" data that we used for evaluating a model fit using "training" data. We will now refer to these held-out samples as the **validation** data and this approach as the **validation set approach**. Along the way we'll redefine the notion of a "test" dataset.

To illustrate the use of resampling techniques, we'll consider a regression setup with a single predictor x, and a regression function $f(x) = x^3$. Adding an additional noise parameter, we define the entire data generating process as

$$Y \sim N(\mu = x^3, \sigma^2 = 0.25^2)$$

We write an `R` function that generates datasets according to this process.

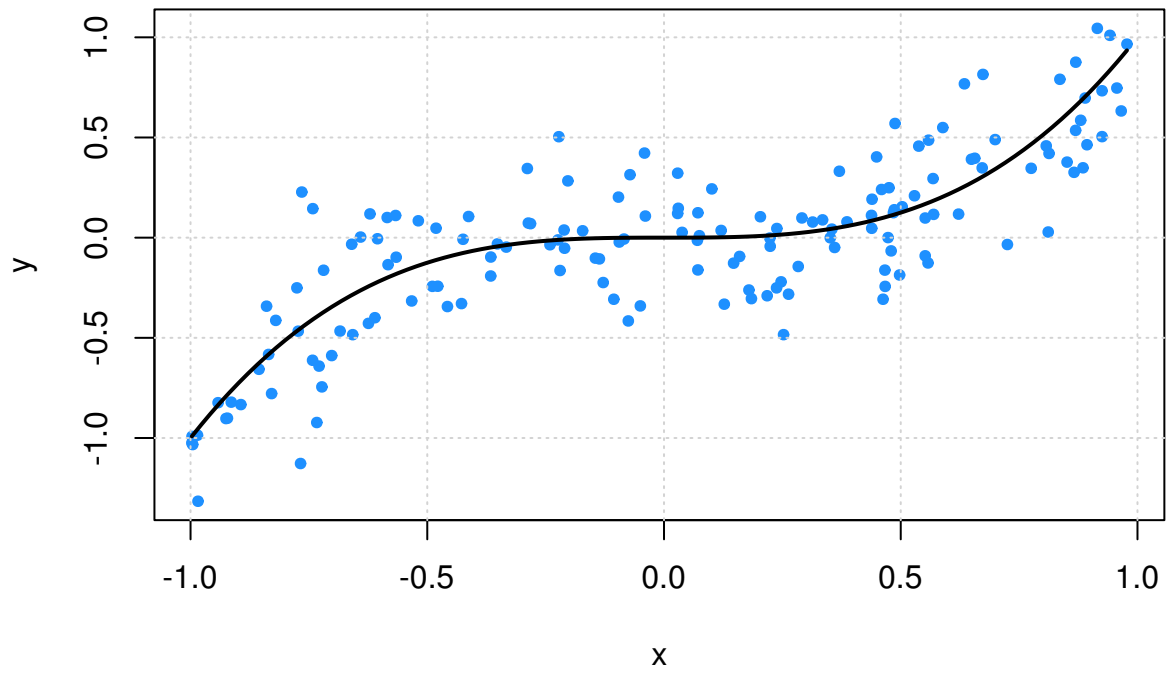
```
gen_sim_data = function(sample_size) {
  x = runif(n = sample_size, min = -1, max = 1)
  y = rnorm(n = sample_size, mean = x ^ 3, sd = 0.25)
  data.frame(x, y)
}
```

We first simulate a single dataset, which we also split into a *train* and *validation* set. Here, the validation set is 20% of the data.

```
set.seed(42)
sim_data = gen_sim_data(sample_size = 200)
sim_idx  = sample(1:nrow(sim_data), 160)
sim_trn  = sim_data[sim_idx, ]
sim_val  = sim_data[-sim_idx, ]
```

We plot this training data, as well as the true regression function.

```
plot(y ~ x, data = sim_trn, col = "dodgerblue", pch = 20)
grid()
curve(x ^ 3, add = TRUE, col = "black", lwd = 2)
```




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

Recall that we needed this validation set because the training error was far too optimistic for highly flexible models. This would lead us to always use the most flexible model.

```
fit = lm(y ~ poly(x, 10), data = sim_trn)

calc_rmse(actual = sim_trn$y, predicted = predict(fit, sim_trn))
```

```
## [1] 0.2262618
```

```
calc_rmse(actual = sim_val$y, predicted = predict(fit, sim_val))
```

```
## [1] 0.2846442
```

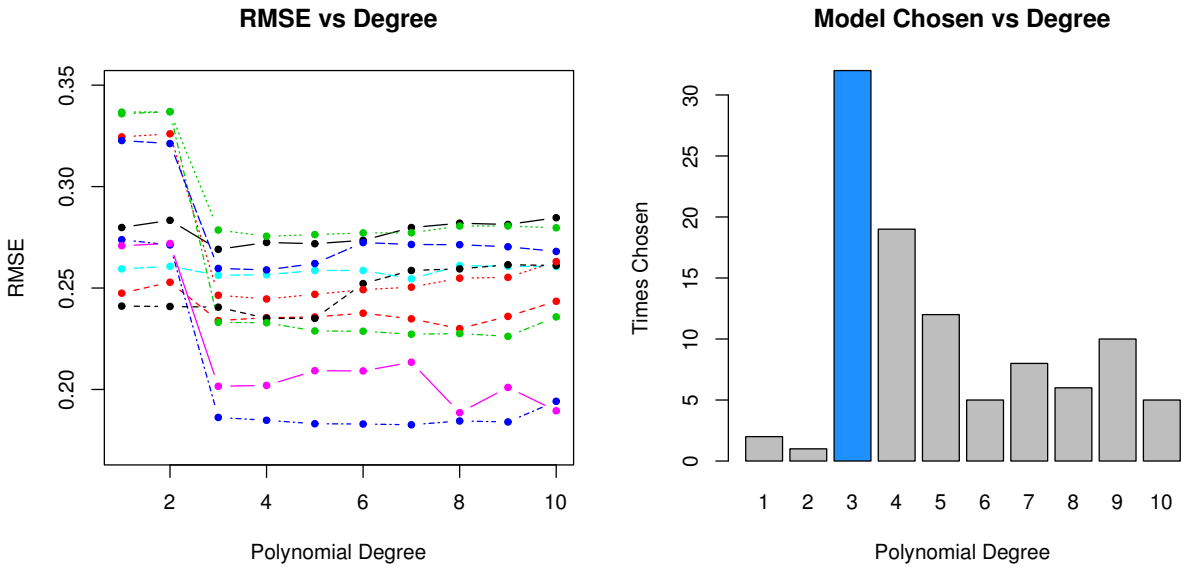
20.1 Validation-Set Approach

- TODO: consider fitting polynomial models of degree k = 1:10 to data from this data generating process
- TODO: here, we can consider k, the polynomial degree, as a tuning parameter
- TODO: perform simulation study to evaluate how well validation set approach works

```
num_sims = 100
num_degrees = 10
val_rmse = matrix(0, ncol = num_degrees, nrow = num_sims)
```

- TODO: each simulation we will...

```
set.seed(42)
for (i in 1:num_sims) {
  # simulate data
  sim_data = gen_sim_data(sample_size = 200)
  # set aside validation set
  sim_idx = sample(1:nrow(sim_data), 160)
  sim_trn = sim_data[sim_idx, ]
  sim_val = sim_data[-sim_idx, ]
  # fit models and store RMSEs
  for (j in 1:num_degrees) {
    #fit model
    fit = glm(y ~ poly(x, degree = j), data = sim_trn)
    # calculate error
    val_rmse[i, j] = calc_rmse(actual = sim_val$y, predicted = predict(fit, sim_val))
  }
}
```



- TODO: issues are hard to "see" but have to do with variability

20.2 Cross-Validation

Instead of using a single test-train split, we instead look to use K-fold cross-validation.

$$\text{RMSE-CV}_K = \sum_{k=1}^{K} \frac{n_k}{n} \text{RMSE}_k$$

$$\text{RMSE}_k = \sqrt{\frac{1}{n_k} \sum_{i \in C_k} \left(y_i - \hat{f}^{-k}(x_i) \right)^2}$$

- n_k is the number of observations in fold k
- C_k are the observations in fold k
- $\hat{f}^{-k}()$ is the trained model using the training data without fold k

If n_k is the same in each fold, then

$$\text{RMSE-CV}_K = \frac{1}{K}\sum_{k=1}^{K}\text{RMSE}_k$$

- TODO: create and add graphic that shows the splitting process
- TODO: Can be used with any metric, MSE, RMSE, class-err, class-acc

There are many ways to perform cross-validation in `R`, depending on the statistical learning method of interest. Some methods, for example `glm()` through `boot::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(y ~ poly(x, 3), data = sim_trn)
coef(glm_fit)
```

```
##  (Intercept)  poly(x, 3)1  poly(x, 3)2  poly(x, 3)3
## -0.005513063  4.153963639 -0.207436179  2.078844572
```

```
lm_fit  = lm(y ~ poly(x, 3), data = sim_trn)
coef(lm_fit)
```

```
##  (Intercept)  poly(x, 3)1  poly(x, 3)2  poly(x, 3)3
## -0.005513063  4.153963639 -0.207436179  2.078844572
```

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

```
sqrt(boot::cv.glm(sim_trn, glm_fit)$delta)
```

```
## [1] 0.2372763 0.2372582
```

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

In practice, we often prefer 5 or 10-fold cross-validation for a number of reason, but often most importantly, for computational efficiency.

```
sqrt(boot::cv.glm(sim_trn, glm_fit, K = 5)$delta)
```

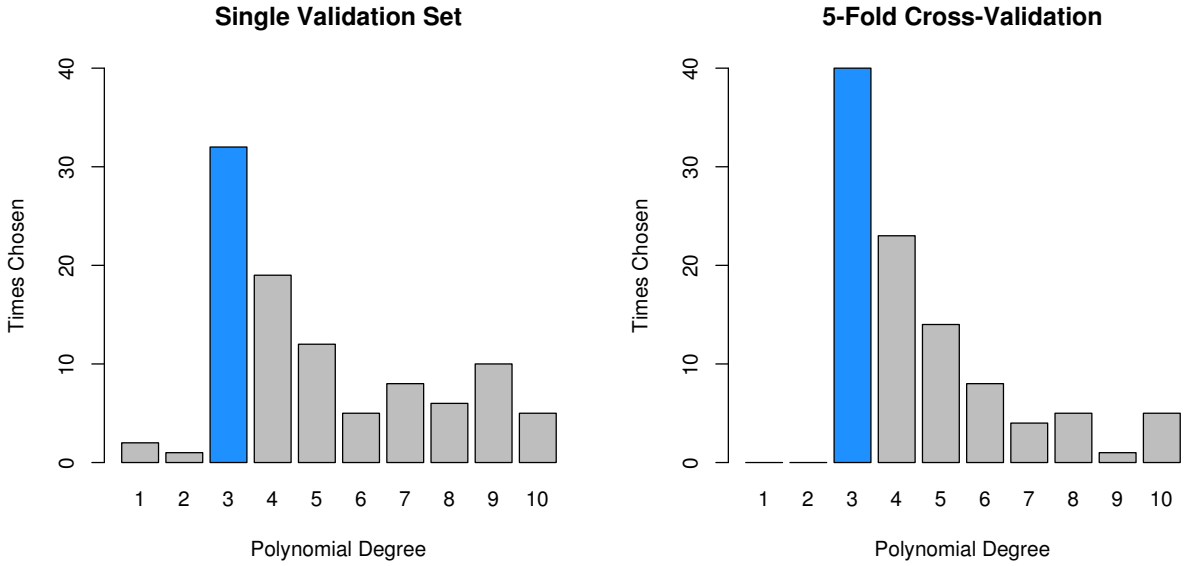
```
## [1] 0.2392979 0.2384206
```

We repeat the above simulation study, this time performing 5-fold cross-validation. With a total sample size of $n = 200$ each validation set has 40 observations, as did the single validation set in the previous simulations.

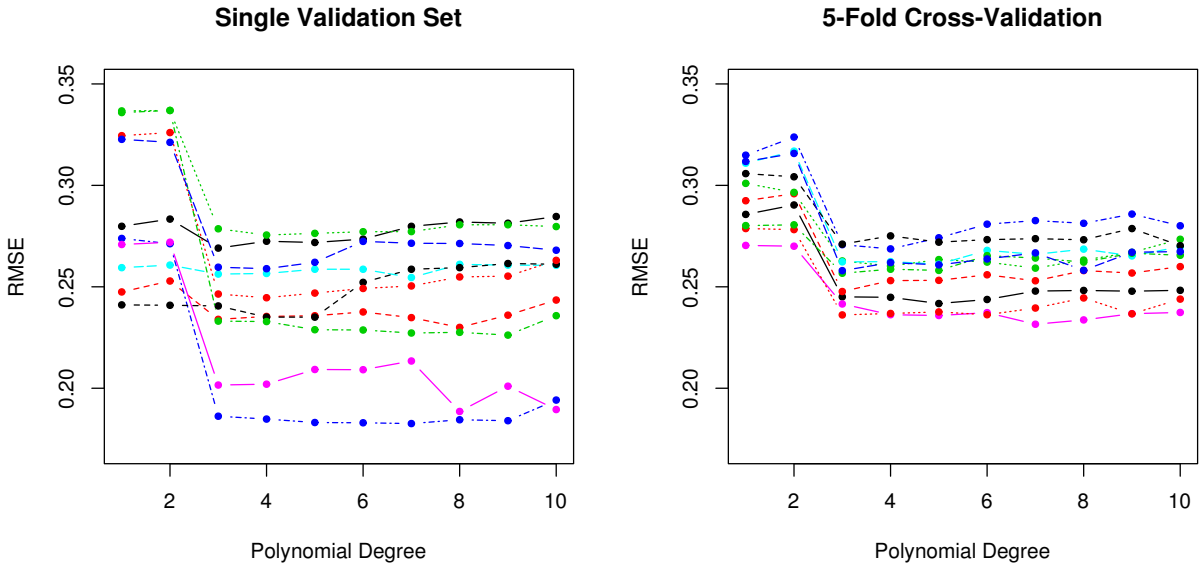
```
cv_rmse = matrix(0, ncol = num_degrees, nrow = num_sims)
```

```
set.seed(42)
for (i in 1:num_sims) {
  # simulate data, use all data for training
  sim_trn = gen_sim_data(sample_size = 200)
  # fit models and store RMSE
  for (j in 1:num_degrees) {
    #fit model
    fit = glm(y ~ poly(x, degree = j), data = sim_trn)
    # calculate error
    cv_rmse[i, j] = sqrt(boot::cv.glm(sim_trn, fit, K = 5)$delta[1])
```

```
  }
}
```



Polynomial Degree	Mean, Val	SD, Val	Mean, CV	SD, CV
1	0.292	0.031	0.294	0.015
2	0.293	0.031	0.295	0.015
3	0.252	0.028	0.255	0.012
4	0.253	0.028	0.255	0.013
5	0.254	0.028	0.256	0.013
6	0.254	0.028	0.257	0.013
7	0.255	0.028	0.258	0.013
8	0.256	0.029	0.258	0.013
9	0.257	0.029	0.261	0.013
10	0.259	0.030	0.262	0.014



- TODO: differences: less variance, better selections

20.3 Test Data

The following example, inspired by The Elements of Statistical Learning, will illustrate the need for a dedicated test set which is **never** used in model training. We do this, if for no other reason, because it gives us a quick sanity check that we have cross-validated correctly. To be specific we will always test-train split the data, then perform cross-validation **within the training data**.

Essentially, this example will also show how to **not** cross-validate properly. It will also show can example of cross-validated in a classification setting.

```
calc_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

Consider a binary response Y with equal probability to take values 0 and 1.

$$Y \sim \text{bern}(p = 0.5)$$

Also consider $p = 10,000$ independent predictor variables, X_j, each with a standard normal distribution.

$$X_j \sim N(\mu = 0, \sigma^2 = 1)$$

We simulate $n = 100$ observations from this data generating process. Notice that the way we've defined this process, none of the X_j are related to Y.

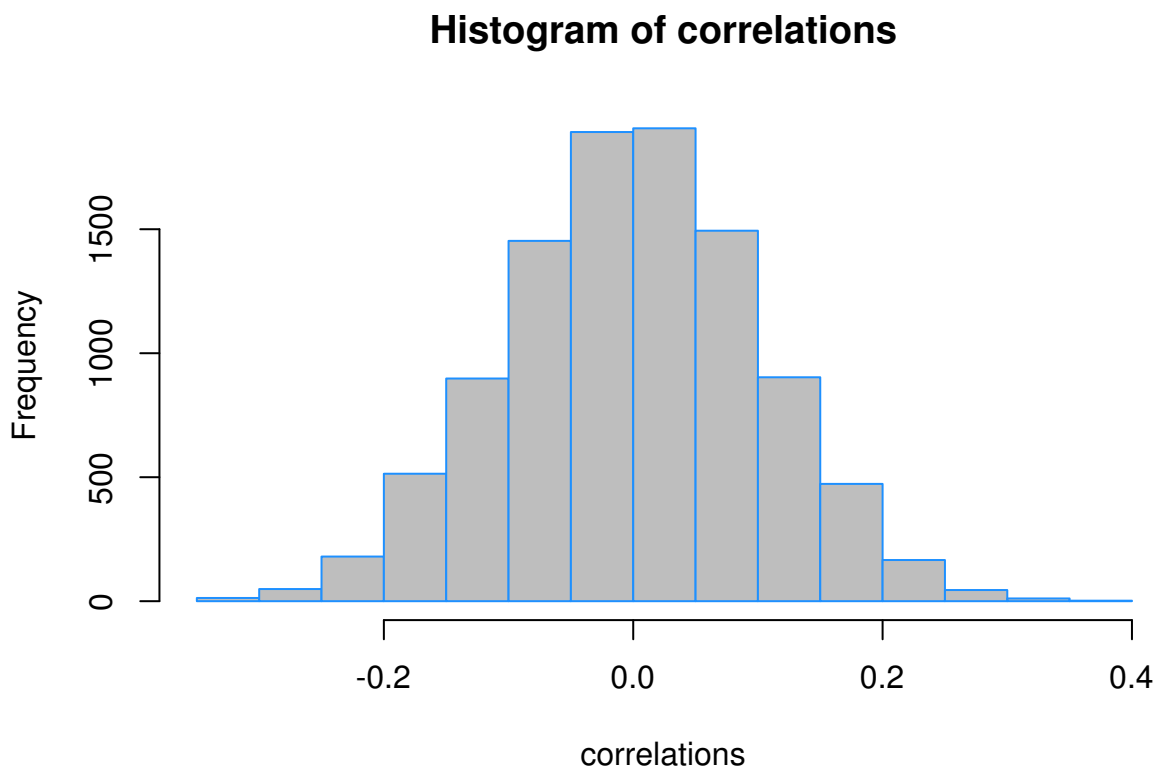
```
set.seed(42)
n = 200
p = 10000
x = replicate(p, rnorm(n))
y = c(rbinom(n = n, size = 1, prob = 0.5))
full_data = data.frame(y, x)
```

Before attempting to perform cross-validation, we test-train split the data, using half of the available data for each. (In practice, with this little data, it would be hard to justify a separate test dataset, but here we do so to illustrate another point.)

```
trn_idx  = sample(1:nrow(full_data), trunc(nrow(full_data) * 0.5))
trn_data = full_data[trn_idx,   ]
tst_data = full_data[-trn_idx, ]
```

Now we would like to train a logistic regression model to predict Y using the available predictor data. However, here we have $p > n$, which prevents us from fitting logistic regression. To overcome this issue, we will first attempt to find a subset of relevant predictors. To do so, we'll simply find the predictors that are most correlated with the response.

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



While many of these correlations are small, many very close to zero, some are as large as 0.40. Since our training data has 50 observations, we'll select the 25 predictors with the largest (absolute) correlations.

```
selected = order(abs(correlations), decreasing = TRUE)[1:25]
correlations[selected]
```

```
##      X2596      X4214      X9335      X8569      X3299      X2533
##  0.3543596  0.3523432 -0.3479568 -0.3457459 -0.3454538  0.3432992
##      X2638      X4737      X2542      X8624      X6201      X4186
## -0.3393733 -0.3314835  0.3228942 -0.3193488  0.3187754 -0.3181454
##      X7600      X8557      X3273      X5639      X4482      X7593
##  0.3175957  0.3159638 -0.3117192  0.3113686  0.3109364  0.3094102
##      X7374      X7283      X9888       X518      X9970      X7654
```

```
##  0.3090942 -0.3086637  0.3069136 -0.3066874 -0.3061039 -0.3042648
##      X9329
## -0.3038140
```

We subset the training and test sets to contain only the response as well as these 25 predictors.

```
trn_screen = trn_data[c(1, selected)]
tst_screen = tst_data[c(1, selected)]
```

Then we finally fit an additive logistic regression using this subset of predictors. We perform 10-fold cross-validation to obtain an estimate of the classification error.

```
add_log_mod = glm(y ~ ., data = trn_screen, family = "binomial")
boot::cv.glm(trn_screen, add_log_mod, K = 10)$delta[1]
```

```
## [1] 0.3166792
```

The 10-fold cross-validation is suggesting a classification error estimate of almost 30%.

```
add_log_pred = (predict(add_log_mod, newdata = tst_screen, type = "response") > 0.5) * 1
calc_err(predicted = add_log_pred, actual = tst_screen$y)
```

```
## [1] 0.5
```

However, if we obtain an estimate of the error using the set, we see an error rate of 50%. No better than guessing! But since Y has no relationship with the predictors, this is actually what we would expect. This incorrect method we'll call screen-then-validate.

Now, we will correctly screen-while-validating. Essentially, instead of simply cross-validating the logistic regression, we also need to cross validate the screening process. That is, we won't simply use the same variables for each fold, we get the "best" predictors for each fold.

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 completely 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(trn_data$y, k = 10)
```

```
## $Fold01
##  [1]  2  6 10 28 66 69 70 89 94 98
##
## $Fold02
##  [1] 27 30 32 33 34 56 74 80 85 96
##
## $Fold03
##  [1]  8 23 29 31 39 53 57 60 61 72
##
## $Fold04
##  [1]  9 15 16 21 41 44 54 63 71 99
##
## $Fold05
##  [1]  5 12 17 51 62 68 81 82 92 97
##
## $Fold06
##  [1]  7 13 19 40 43 55 75 77 87 90
##
## $Fold07
```

```
##  [1]  18  42  45  47  48  73  83  88  91 100
##
## $Fold08
##  [1]  4 11 35 37 46 52 64 76 79 84
##
## $Fold09
##  [1]  1 14 20 22 26 36 50 59 67 78
##
## $Fold10
##  [1]  3 24 25 38 49 58 65 86 93 95
```

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

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

for (i in seq_along(folds)) {

  # split for fold i
  trn_fold = trn_data[-folds[[i]], ]
  val_fold = trn_data[folds[[i]], ]

  # screening for fold i
  correlations = apply(trn_fold[, -1], 2, cor, y = trn_fold[,1])
  selected = order(abs(correlations), decreasing = TRUE)[1:25]
  trn_fold_screen = trn_fold[ , c(1, selected)]
  val_fold_screen = val_fold[ , c(1, selected)]

  # error for fold i
  add_log_mod = glm(y ~ ., data = trn_fold_screen, family = "binomial")
  add_log_prob = predict(add_log_mod, newdata = val_fold_screen, type = "response")
  add_log_pred = ifelse(add_log_prob > 0.5, yes = 1, no = 0)
  fold_err[i] = mean(add_log_pred != val_fold_screen$y)

}

# report all 10 validation fold errors
fold_err
```

```
##  [1] 0.5 0.5 0.6 0.5 0.6 0.5 0.7 0.6 0.2 0.2
```

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

```
## [1] 0.49
```

- TODO: note that, even cross-validated correctly, this isn't a brilliant variable selection procedure. (it completely ignores interactions and correlations among the predictors. however, if it works, it works.) next chapters...

20.4 Bootstrap

ISL discusses the bootstrap, which is another resampling method. However, it is less relevant to the statistical learning tasks we will encounter. It could be used to replace cross-validation, but encounters significantly more computation.

It could be more 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.

20.5 Which K?

- TODO: LOO vs 5 vs 10
- TODO: bias and variance

20.6 Summary

- TODO: using cross validation for: tuning, error estimation

20.7 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.

20.8 `rmarkdown`

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1.

Chapter 21

The `caret` Package

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:

- Test-train split the available data
  - Consider a method
    - Decide on a set of candidate models (specify possible tuning parameters for method)
    - Use resampling to find the "best model" by choosing the values of the tuning parameters
  - 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. Some methods cannot handle factor variables.

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.
- Specify possible tuning parameters for method
  - `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.
- Use resampling to find the "best model" by choosing the values of the tuning parameters
  - `trainControl()` will specify the resampling scheme
  - `train()` is the workhorse of `caret`. It takes the following information then trains (tunes) the requested model:
    - `form`, a formula, such as `y ~ .`
      - This specifies the response and which predictors (or transformations of) should be used.
    - `data`, the data used for training
    - `trControl` which specifies the resampling scheme, that is, how cross-validation should be performed to find the best values of the tuning parameters
    - `preProcess` which allows for specification of data pre-processing such as centering and scaling

    * `method`, a statistical learning method from a long list of availible models
    * `tuneGrid` which specifies the tuning parameters to train over
- Use chosen model to make predictions
  - `predict()` used on objects of type `train` will be truly magical!

21.1 Classification

To illustrate `caret`, first for classification, we will use the `Default` data from the `ISLR` package.

```
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, ]
```

At first glance, it might appear as if the use of `createDataPartition()` is no different than our previous use of `sample()`. However, `createDataPartition()` tries to ensure a split that has a similar distribution of the supplied variable in both datasets. See the documentation for details.

After splitting the data, we can begin training a number of models. We begin with a simple additive logistic regression.

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

Here, we have supplied four arguments to the `train()` function form the `caret` package.

- `form = default ~ .` specifies the `default` variable as the response. It also indicates that all available predictors should be used.
- `data = default_trn` specifies that training will be down with the `default_trn` data
- `trControl = trainControl(method = "cv", number = 5)` specifies that we will be using 5-fold cross-validation.
- `method = glm` specifies that we will fit a generalized linear model.

The `method` essentially specifies both the model (and more specifically the function to fit said model in `R`) and package that will be used. The `train()` function is essentially a wrapper around whatever method we chose. In this case, the function is the base `R` function `glm()`, so no additional package is required. When a method requires a function from a certain package, that package will need to be installed. See the list of availible models for package information.

The list that we have passed to the `trControl` argument is created using the `trainControl()` function from `caret`. The `trainControl()` function is a powerful tool for specifying a number of the training choices required by `train()`, in particular the resampling scheme.

```
trainControl(method = "cv", number = 5)[1:3]
```

```
## $method
## [1] "cv"
```

```
##
## $number
## [1] 5
##
## $repeats
## [1] NA
```

Here we see just the first three elements of this list, which are related to how the resampling will be done. These are the three elements that we will be most interested in. Here, only the first two are relevant.

- `method` specifies how resampling will be done. Examples include `cv`, `boot`, `LOOCV`, `repeatedcv`, and `oob`.
- `number` specifies the number of times resampling should be done for methods that require resample, such as, `cv` and `boot`.
- `repeats` specifies the number of times to repeat resampling for methods such as `repeatedcv`

For details on the full capabilities of this function, see the relevant documentation. The out-of-bag, `oob` which is a sort of automatic resampling for certain statistical learning methods, will be introduced later.

We've also passed an additional argument of `"binomial"` to `family`. This isn't actually an argument for `train()`, but an additional argument for the method `glm`. In actuality, we don't need to specify the family. Since `default` is a factor variable, `caret` automatically detects that we are trying to perform classification, and would automatically use `family = "binomial"`. This isn't the case if we were simply using `glm()`.

```
default_glm_mod
```

```
## 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
```

Called the stored `train()` object summarizes the training that we have done. We see that we used 7501 observations that had a binary class response and three predictors. We have not done any data pre-processing, and have utilized 5-fold cross-validation. The cross-validated accuracy is reported. Note that, `caret` is an optimist, and prefers to report accuracy (proportion of correct classifications) instead of the error that we often considered before (proportion of incorrect classifications).

```
names(default_glm_mod)
```

```
##  [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"
```

We see that there is a wealth of information stored in the list returned by `train()`. Two elements that we will often be interested in are `results` and `finalModel`.

```
default_glm_mod$results
```

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

The `resutls` show some more detailed results, in particular `AccuracySD` which gives us an estimate of the uncertainty in our accuracy estimate.

```
default_glm_mod$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
```

The `finalModel` is a model object, in this case, the object returned from `glm()`. This final model, is fit to all of the supplied training data. This model object is often used when we call certain relevant functions on the object returned by `train()`, such as `summary()`

```
summary(default_glm_mod)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4350  -0.1486  -0.0588  -0.0218   3.7184
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.066e+01  5.509e-01 -19.353   <2e-16 ***
## studentYes  -6.254e-01  2.702e-01  -2.315   0.0206 *
## balance      5.647e-03  2.639e-04  21.401   <2e-16 ***
## income       1.395e-06  9.300e-06   0.150   0.8808
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2192.2  on 7500  degrees of freedom
## Residual deviance: 1203.5  on 7497  degrees of freedom
## AIC: 1211.5
##
## Number of Fisher Scoring iterations: 8
```

We see that this summary is what we had seen previously from objects of type `glm`.

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

To obtain test accuracy, we will need to make predictions on the test data. With the object returned by `train()`, this is extremely easy.

```
head(predict(default_glm_mod, newdata = default_tst))
```

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

We see that by default, the `predict()` function is returning classifications. This will be true no matter what method we use!

```
# test acc
calc_acc(actual = default_tst$default,
         predicted = predict(default_glm_mod, newdata = default_tst))
```

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

If instead of the default behavior of returning classifications, we instead wanted predicted probabilities, we simply specify `type = "prob"`.

```
# get probs
head(predict(default_glm_mod, 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
```

Notice that this returns the probabilities for all possible classes, in this case `No` and `Yes`. Again, this will be true for all methods! This is especially useful for multi-class data!.

21.1.1 Tuning

Since logistic regression has no tuning parameters, we haven't really highlighted the full potential of `caret`. We've essentially used it to obtain cross-validated results, and for the more well-behaved `predict()` function. These are excellent improvements over our previous methods, but the real power of `caret` is its ability to provide a framework for tuning model.

To illustrate tuning, we now use `knn` as our method, which performs k-nearest neighbors.

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

First, note that we are using formula syntax here, where previously we needed to create separate response and predictors matrices. Also, we're using a factor variable as a predictor, and `caret` seems to be taking care of this automatically.

```
default_knn_mod
```

```
## 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.
```

Here we are again using 5-fold cross-validation and no pre-processing. Notice that we now have multiple results, for `k = 5`, `k = 7`, and `k = 9`.

Let's modifying this training by introducing pre-processing, and specifying our own tuning parameters, instead of the default values above.

```
default_knn_mod = train(
  default ~ .,
  data = default_trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  preProcess = c("center", "scale"),
  tuneGrid = expand.grid(k = seq(1, 101, by = 2))
)
```

Here, we've specified that we would like to center and scale the data. Essentially transforming each predictor to have mean 0 and variance 1. The documentation on the `preProcess()` function provides examples of additional possible pre-processing. IN our call to `train()` we're essentially specifying how we would like this function applied to our data.

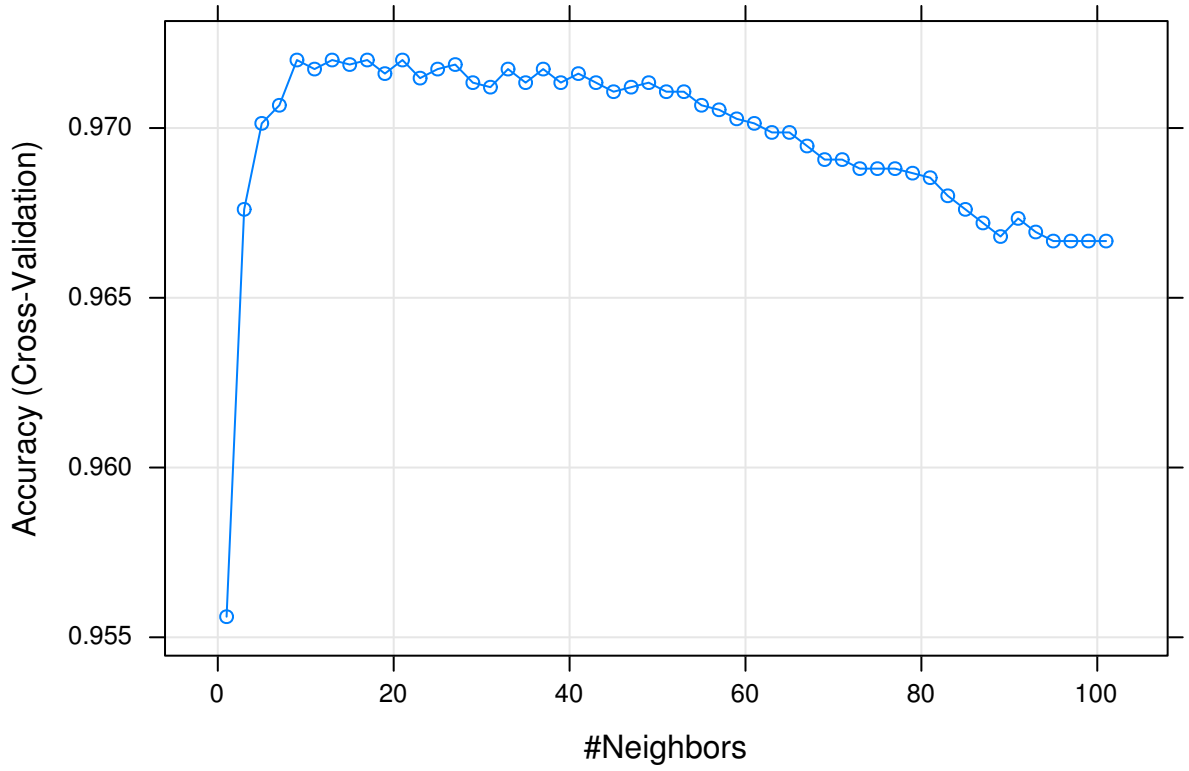
We've also provided a "tuning grid," in this case, the values of `k` to try. The `tuneGrid` argument expects a data frame, which `expand.grid()` returns. We don't actually need `expand.grid()` for this example, but it will be a useful habit to develop when we move to methods with multiple tuning parameters.

```
head(default_knn_mod$results, 5)
```

```
##   k  Accuracy     Kappa  AccuracySD    KappaSD
## 1 1 0.9556062 0.2980896 0.003446167 0.04848084
## 2 3 0.9676048 0.3753023 0.002959213 0.04629315
## 3 5 0.9701374 0.4040223 0.001442498 0.04092678
## 4 7 0.9706704 0.3906642 0.001567466 0.05200146
## 5 9 0.9720034 0.4091064 0.001706297 0.05361118
```

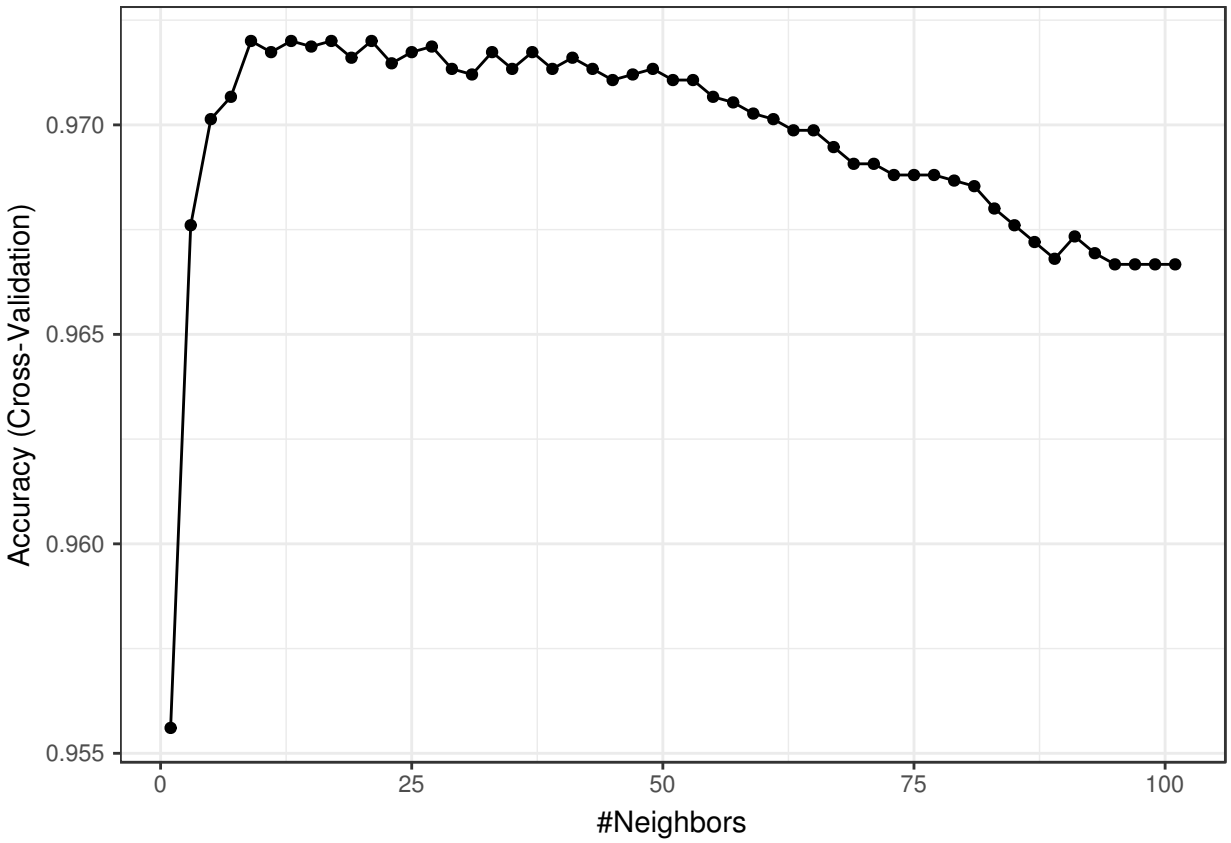
Since how we have a large number of results, display the entire `results` would create a lot of clutter. Instead, we can plot the tuning results by calling `plot()` on the object returned by `train()`.

```
plot(default_knn_mod)
```



By default, `caret` utilizes the `lattice` graphics package to create these plots. Recently, additional support for `ggplot2` style graphics has been added for some plots.

```
ggplot(default_knn_mod) + theme_bw()
```



Now that we are dealing with a tuning parameter, `train()` determines the best value of those considered, by default selecting the best (highest cross-validated) accuracy, and returning that value as `bestTune`.

```
default_knn_mod$bestTune
```

```
##    k
## 9 17
```

```
get_best_result = function(caret_fit) {
  best = which(rownames(caret_fit$results) == rownames(caret_fit$bestTune))
  best_result = caret_fit$results[best, ]
  rownames(best_result) = NULL
  best_result
}
```

Sometimes it will be useful to obtain the results for only that value. The above function does this automatically.

```
get_best_result(default_knn_mod)
```

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

While we did fit a large number of models, the "best" model is stored in `finalModel`. After this model was determined to be the best via cross-validation, it is then fit to the entire training dataset.

```
default_knn_mod$finalModel
```

```
## 17-nearest neighbor model
## Training set outcome distribution:
```

```
##
##   No  Yes
## 7251  250
```

With this model, we can again make predictions and obtain predicted probabilities.

```
head(predict(default_knn_mod, newdata = default_tst, type = "prob"))
```

```
##          No        Yes
## 1 1.0000000 0.00000000
## 2 1.0000000 0.00000000
## 3 1.0000000 0.00000000
## 4 0.9411765 0.05882353
## 5 1.0000000 0.00000000
## 6 1.0000000 0.00000000
```

As an example of a multi-class response consider the following three models fit to the the `iris` data. Note that the first model is essentially "multinomial logistic regression," but you might notice it also has a tuning parameter now. (Spoiler: It's actually a neural network, so you'll need the `nnet` package.)

```
iris_log_mod = train(
  Species ~ .,
  data = iris,
  method = "multinom",
  trControl = trainControl(method = "cv", number = 5),
  trace = FALSE
)
```

```
iris_knn_mod = train(
  Species ~ .,
  data = iris,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  preProcess = c("center", "scale"),
  tuneGrid = expand.grid(k = seq(1, 21, by = 2))
)
```

```
iris_qda_mod = train(
  Species ~ .,
  data = iris,
  method = "qda",
  trControl = trainControl(method = "cv", number = 5)
)
```

We can obtain predicted probabilities with these three models. Notice that they give the predicted probability for **each** class, using the same syntax for each model.

```
head(predict(iris_log_mod, type = "prob"))
```

```
##      setosa  versicolor    virginica
## 1 0.9884905 0.011509549 6.415435e-10
## 2 0.9637873 0.036212696 8.263511e-09
## 3 0.9804027 0.019597292 3.376946e-09
## 4 0.9568631 0.043136865 2.083006e-08
## 5 0.9903391 0.009660933 5.465878e-10
## 6 0.9907892 0.009210789 8.230133e-10
```

```
head(predict(iris_knn_mod, type = "prob"))
```

```
##   setosa versicolor virginica
## 1      1          0         0
## 2      1          0         0
## 3      1          0         0
## 4      1          0         0
## 5      1          0         0
## 6      1          0         0
```

```
head(predict(iris_qda_mod, type = "prob"))
```

```
##   setosa   versicolor    virginica
## 1      1 4.918517e-26 2.981541e-41
## 2      1 7.655808e-19 1.311032e-34
## 3      1 1.552279e-21 3.380440e-36
## 4      1 8.300396e-19 8.541858e-32
## 5      1 3.365614e-27 2.010147e-41
## 6      1 1.472533e-26 1.271928e-40
```

21.2 Regression

To illustrate the use of `caret` for regression, we'll consider some simulated data.

```
gen_some_data = function(n_obs = 50) {
  x1 = seq(0, 10, length.out = n_obs)
  x2 = runif(n = n_obs, min = 0, max = 2)
  x3 = sample(c("A", "B", "C"), size = n_obs, replace = TRUE)
  x4 = round(runif(n = n_obs, min = 0, max = 5), 1)
  x5 = round(runif(n = n_obs, min = 0, max = 5), 0)
  y = round(x1 ^ 2 + x2 ^ 2 + 2 * (x3 == "B") + rnorm(n = n_obs), 3)
  data.frame(y, x1, x2, x3, x4, x5)
}
```

We first simulate a train and test dataset.

```
set.seed(42)
sim_trn = gen_some_data(n_obs = 500)
sim_tst = gen_some_data(n_obs = 5000)
```

Fitting `knn` works nearly identically to its use for classification. Really, the only difference here is that we have a numeric response, which `caret` understands to be a regression problem.

```
sim_knn_mod = train(
  y ~ .,
  data = sim_trn,
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  # preProcess = c("center", "scale"),
  tuneGrid = expand.grid(k = seq(1, 31, by = 2))
)
```

```
sim_knn_mod$modelType
```

```
## [1] "Regression"
```

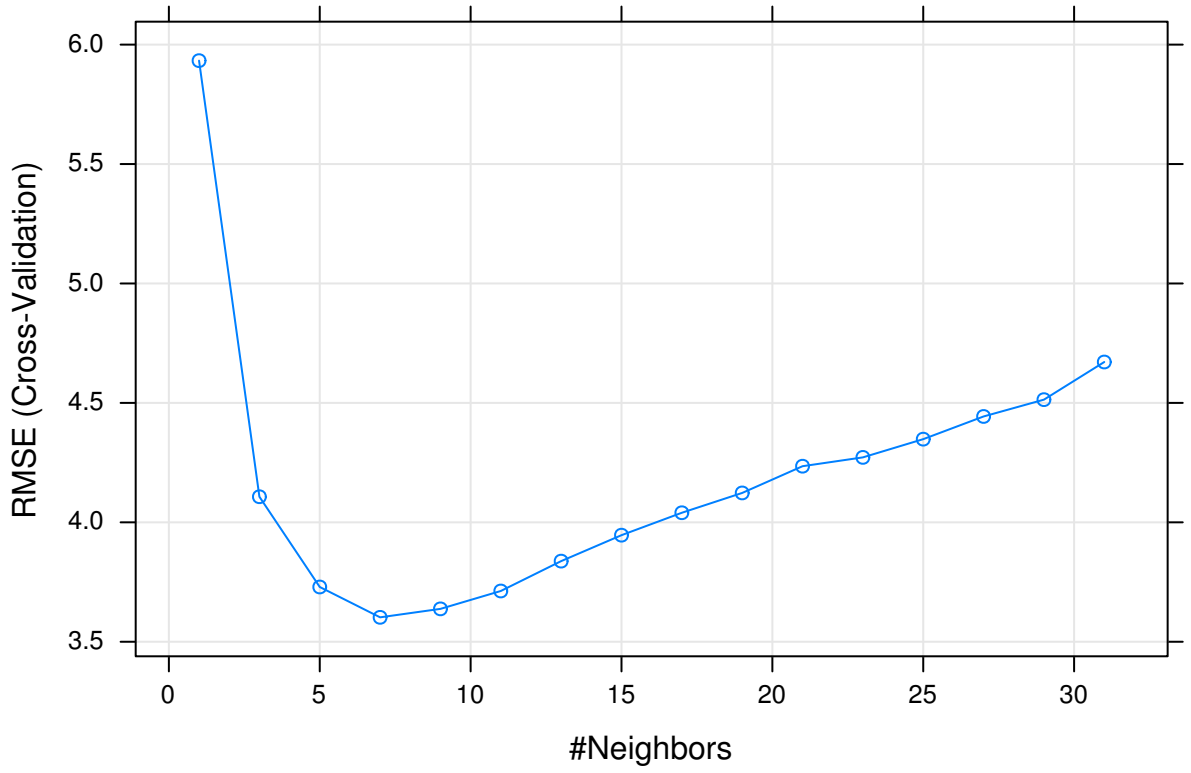
Notice that we've commented out the line to perform pre-processing. Can you figure out why?

```
get_best_result(sim_knn_mod)
```

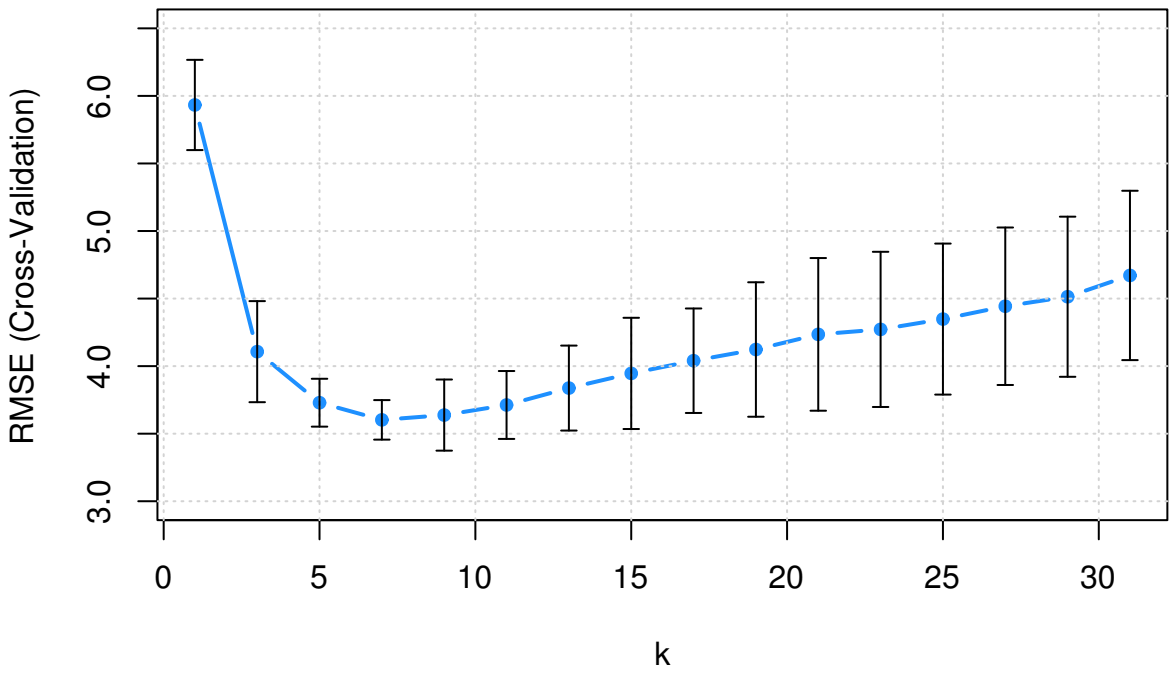
```
##   k     RMSE  Rsquared      MAE    RMSESD   RsquaredSD     MAESD
## 1 7 3.602252 0.9870044 2.537763 0.1462808 0.0007503021 0.1388759
```

A few things to notice in the results. In addition to the usual RMSE, which is be used to determine the best model, we also have MAE, the mean absolute error. We are also given standard deviations of both of these metrics.

```
plot(sim_knn_mod)
```



The following plot adds error bars to RMSE estimate for each `k`.



Sometimes, instead of simply picking the model with the best RMSE (or accuracy), we pick the simplest model within one standard error of the model with the best RMSE. Here then, we would consider `k = 11` instead of `k = 7` since there isn't a statistically significant difference. This is potentially a very good idea in practice. By picking a a simpler model, we are essentially at less risk of overfitting, especially since in practice, future data may be slightly different than the data that we are training on. If you're trying to win a Kaggle competition, this might not be as useful, since often the test and train data come from the exact same source.

- TODO: additional details about 1-SE rule.

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

Since we simulated this data, we have a rather large test dataset. This allows us to compare our cross-validation error estimate, to an estimate using (an impractically large) test set.

```
get_best_result(sim_knn_mod)$RMSE
```

```
## [1] 3.602252
```

```
calc_rmse(actual = sim_tst$y,
          predicted = predict(sim_knn_mod, sim_tst))
```

```
## [1] 3.446897
```

Here we see that the cross-validated RMSE is a bit of an overestimate, but still rather close to the test error. The real question is, are either of these any good? Is this model predicting well? No! Notice that we simulated this data with an error standard deviation of 1!

21.2.1 Methods

Now that `caret` has given us a pipeline for a predictive analysis, we can very quickly and easily test new methods. For example, in an upcoming chapter we will discuss boosted tree models, but now that we understand how to use `caret`, in order to *use* a boosted tree model, we simply need to know the "method" to do so, which in this case is `gbm`. Beyond knowing that the method exists, we just need to know its tuning parameters, in this case, there are four. We actually could get away with knowing nothing about them, and simply specify a `tuneLength`, then `caret` would automatically try some reasonable values.

Instead, we could read the `caret` documentation to find the tuning parameters, as well as the required packages. For now, we'll simply use the following tuning grid. In later chapters, we'll discuss how these effect the model.

```
gbm_grid = expand.grid(interaction.depth = c(1, 2, 3),
                       n.trees = (1:30) * 100,
                       shrinkage = c(0.1, 0.3),
                       n.minobsinnode = 20)
head(gbm_grid)
```

```
##   interaction.depth n.trees shrinkage n.minobsinnode
## 1                 1     100       0.1             20
## 2                 2     100       0.1             20
## 3                 3     100       0.1             20
## 4                 1     200       0.1             20
## 5                 2     200       0.1             20
## 6                 3     200       0.1             20
```

```
set.seed(42)
sim_gbm_mod = train(
  y ~ .,
  data = sim_trn,
  trControl = trainControl(method = "cv", number = 5),
  method = "gbm",
  tuneGrid = gbm_grid,
  verbose = FALSE
)
```

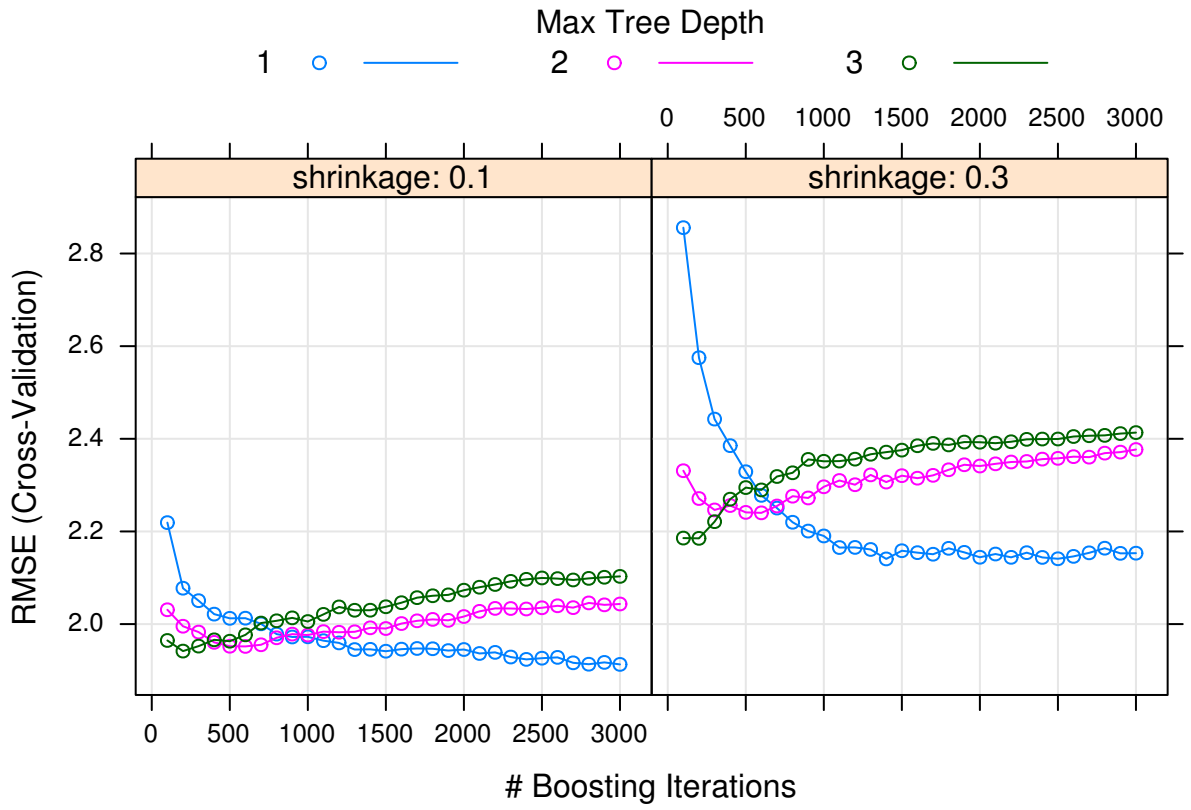
We added `verbose = FALSE` to the `train()` call to suppress some of the intermediate output of the `gbm` fitting procedure.

How this training is happening is a bit of a mystery to us right now. What is this method? How does it deal with the factor variable as a predictor? We'll answer these questions later, for now, we do know how to evaluate how well the method is working.

```
knitr::kable(head(sim_gbm_mod$results), digits = 3)
```

	shrinkage	interaction.depth	n.minobsinnode	n.trees	RMSE	Rsquared	MAE	RMSESD	RsquaredSD
1	0.1	1	20	100	2.219	0.995	1.654	0.214	0.001
91	0.3	1	20	100	2.856	0.991	2.142	0.252	0.002
31	0.1	2	20	100	2.031	0.995	1.424	0.278	0.001
121	0.3	2	20	100	2.331	0.994	1.760	0.168	0.001
61	0.1	3	20	100	1.965	0.996	1.382	0.219	0.001
151	0.3	3	20	100	2.186	0.995	1.613	0.236	0.001

```
plot(sim_gbm_mod)
```




```
sim_gbm_mod$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 30    3000                 1       0.1             20
```

Here we obtain the set of tuning parameters that performed best. Based on the above plot, do you think we considered enough possible tuning parameters?

```
get_best_result(sim_gbm_mod)
```

```
##   shrinkage interaction.depth n.minobsinnode n.trees     RMSE  Rsquared
## 1       0.1                 1             20    3000 1.912962 0.9959445
##        MAE    RMSESD RsquaredSD     MAESD
## 1 1.406154 0.1621041 0.00064417 0.1265028
```

```
calc_rmse(actual = sim_tst$y,
          predicted = predict(sim_gbm_mod, sim_tst))
```

```
## [1] 1.513519
```

Again, the cross-validated result is overestimating the error a bit. Also, this model is a big improvement over the `knn` model, but we can still do better.

```
sim_lm_mod = train(
  y ~ poly(x1, 2) + poly(x2, 2) + x3,
  data = sim_trn,
  method = "lm",
  trControl = trainControl(method = "cv", number = 5)
)
```

```
sim_lm_mod$finalModel
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Coefficients:
##    (Intercept)  `poly(x1, 2)1`  `poly(x1, 2)2`  `poly(x2, 2)1`
##       34.63954       646.63539       166.47950        26.11496
## `poly(x2, 2)2`             x3B             x3C
##        6.77600         1.93871         0.02182
```

Here we fit a good old linear model, except, we specify a very specific formula.

```
sim_lm_mod$results$RMSE
```

```
## [1] 0.9806239
```

```
calc_rmse(actual = sim_tst$y,
          predicted = predict(sim_lm_mod, sim_tst))
```

```
## [1] 1.014825
```

This model dominates the previous two. The `gbm` model does still have a big advantage. The `lm` model needed the correct form of the model, whereas `gbm` nearly learned it automatically!

This question of *which* variables should be included is where we will turn our focus next. We'll consider both what variables are useful for prediction, and learn tools to asses how useful they are.

21.3 External Links

- The `caret` Package - Reference documentation for the `caret` package in `bookdown` format.
- `caret` Model List - List of available models in `caret`.
- `caret` Model List, By Tag - Gives information on tuning parameters and necessary packages.
- Applied Predictive Modeling - Book from the author of the `caret` package, Max Kuhn, as well as Kjell Johnson. Further discussion on use of statistical learning methods in practice.

21.4 `rmarkdown`

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

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)
```

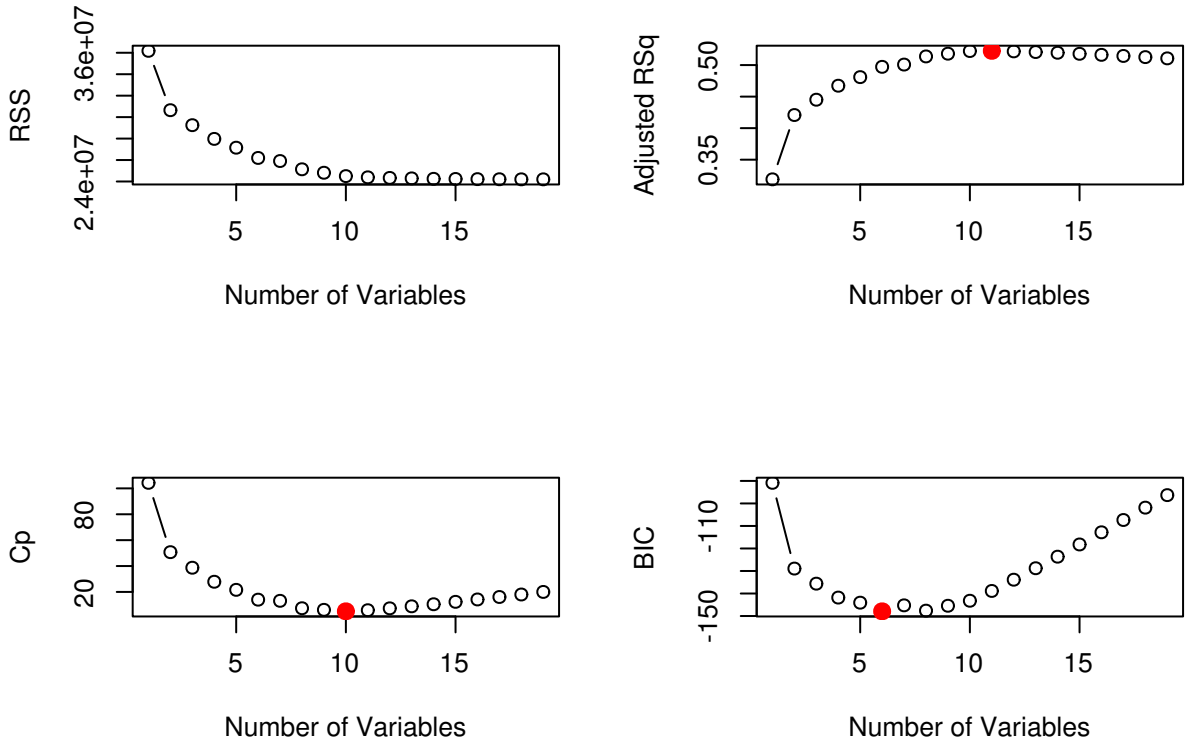
```
## [1] "which"  "rsq"    "rss"    "adjr2"  "cp"     "bic"    "outmat" "obj"
```

```
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
```

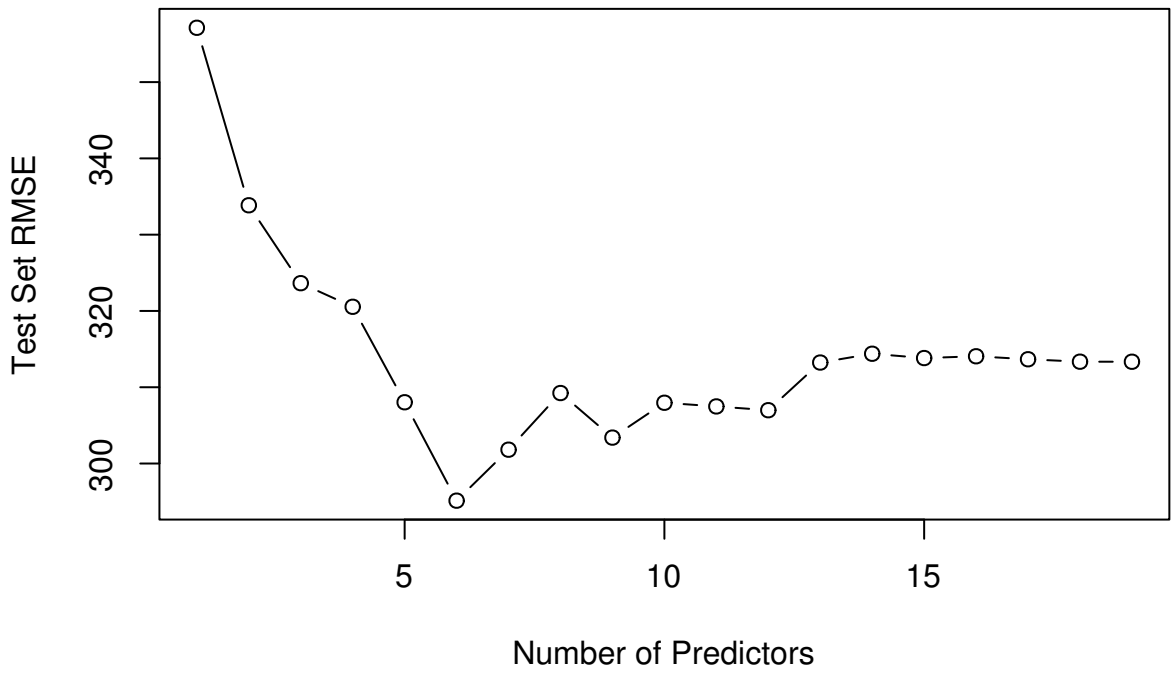
22.2 Validated RMSE

```
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)
```

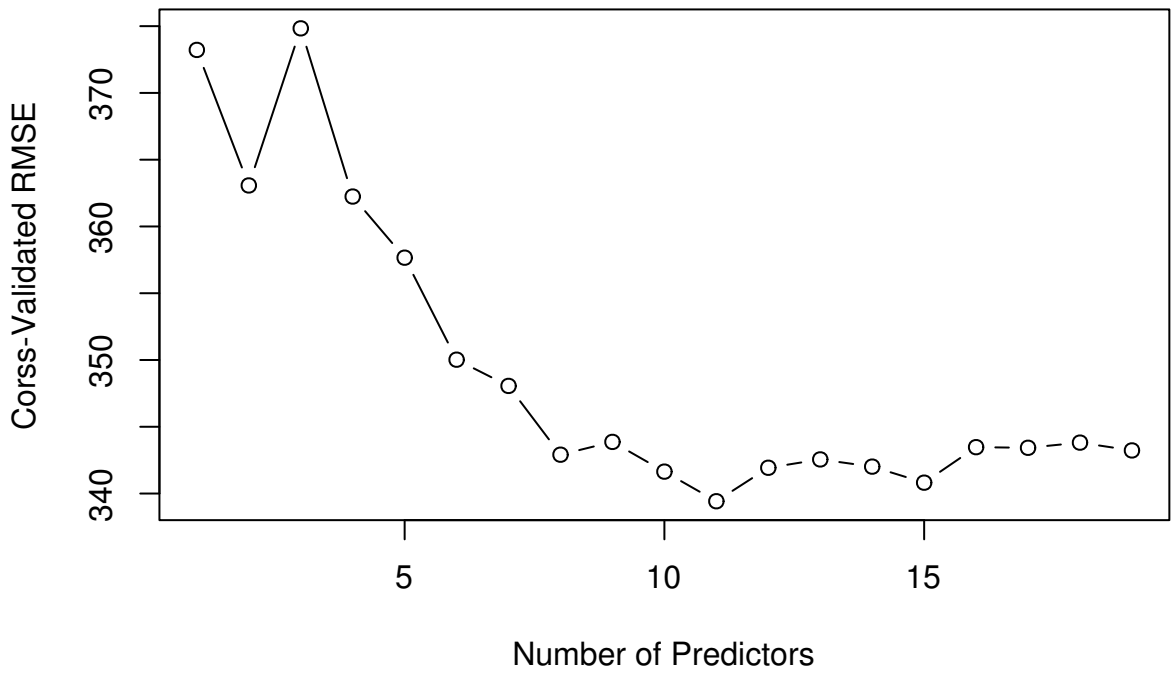
```
  xvars = names(coefs)

  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 3.5.1 and the following packages:

- Base Packages, Attached

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

- Additional Packages, Attached

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

- Additional Packages, Not Attached

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

Part VI

The Modern Era

Chapter 23

Overview

Chapter 24

Regularization

Chapter Status: Currently this chapter is very sparse. It essentially only expands upon an example discussed in ISL, thus only illustrates usage of the methods. Mathematical and conceptual details of the methods will be added later. Also, more comments on using `glmnet` with `caret` will be discussed.

We will use the `Hitters` dataset from the `ISLR` package to explore two shrinkage methods: **ridge regression** 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 from the `glmnet` package to fit penalized regressions.

```
library(glmnet)
```

Unfortunately, the `glmnet` function does not allow the use of model formulas, so we setup the data for ease of use with `glmnet`. Eventually we will use `train()` from `caret` which does allow for fitting penalized regression with the formula syntax, but to explore some of the details, we first work with the functions from `glmnet` directly.

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

First, we fit an ordinary 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

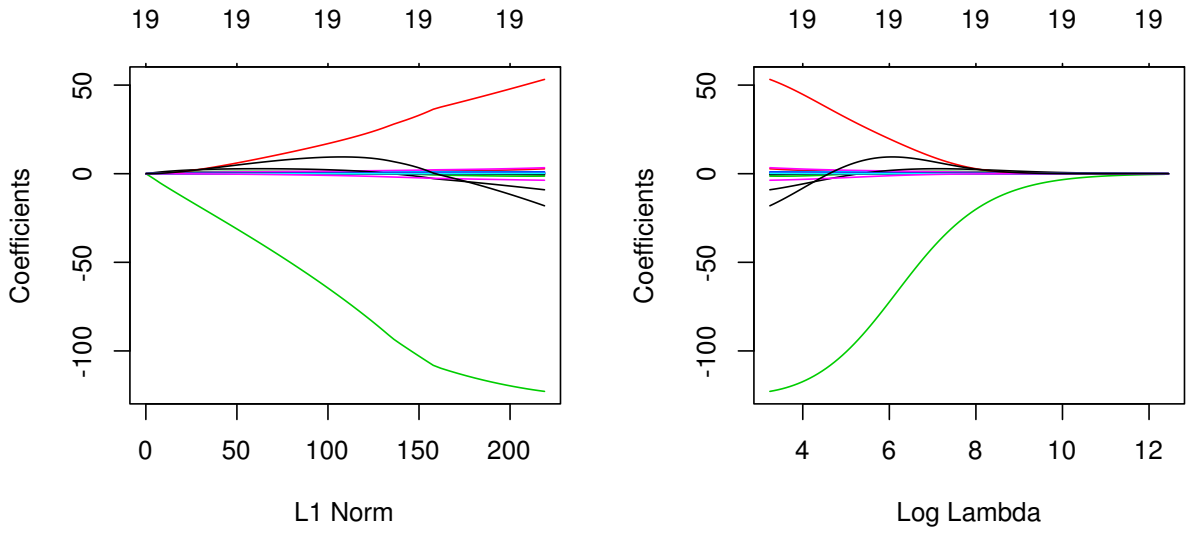
We first illustrate **ridge regression**, which can be fit using `glmnet()` with `alpha = 0` and seeks to minimize

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

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 predictors for fitting, then reports fitted coefficient using the original scale.

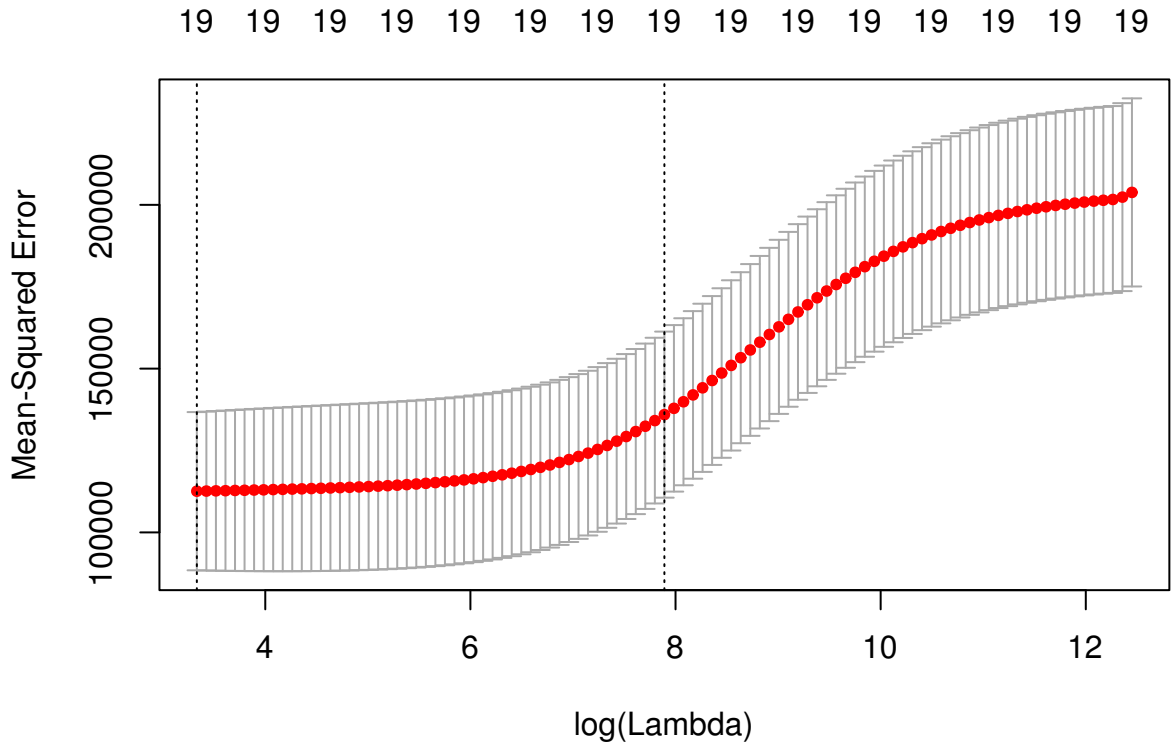
The two plots illustrate how much the coefficients are penalized for different values of λ. Notice none of the coefficients are forced to be zero.

```
par(mfrow = c(1, 2))
fit_ridge = glmnet(X, y, alpha = 0)
plot(fit_ridge)
plot(fit_ridge, xvar = "lambda", label = TRUE)
```



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.)

```
# fitted coefficients, using 1-SE rule lambda, default behavior
coef(fit_ridge_cv)
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                           1
## (Intercept) 213.066444060
## AtBat         0.090095728
## Hits          0.371252755
## HmRun         1.180126954
## Runs          0.596298285
## RBI           0.594502389
## Walks         0.772525465
## Years         2.473494235
## CAtBat        0.007597952
## CHits         0.029272172
## CHmRun        0.217335715
## CRuns         0.058705097
## CRBI          0.060722036
## CWalks        0.058698830
## LeagueN       3.276567808
## DivisionW   -21.889942546
## PutOuts       0.052667119
## Assists       0.007463678
## Errors       -0.145121335
## NewLeagueN    2.972759111
```

```
# fitted coefficients, using minimum lambda
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       1.256355e-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
```

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

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

```
# fitted coefficients, using 1-SE rule lambda
coef(fit_ridge_cv, s = "lambda.1se")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                          1
## (Intercept) 213.066444060
## AtBat         0.090095728
## Hits          0.371252755
## HmRun         1.180126954
## Runs          0.596298285
## RBI           0.594502389
## Walks         0.772525465
## Years         2.473494235
## CAtBat        0.007597952
## CHits         0.029272172
## CHmRun        0.217335715
## CRuns         0.058705097
## CRBI          0.060722036
## CWalks        0.058698830
## LeagueN       3.276567808
## DivisionW   -21.889942546
## PutOuts       0.052667119
## Assists       0.007463678
## Errors       -0.145121335
## NewLeagueN    2.972759111
```

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

```
## [1] 507.788
```

```
# predict using minimum lambda
predict(fit_ridge_cv, X, s = "lambda.min")
```

```
# predict using 1-SE rule lambda, default behavior
predict(fit_ridge_cv, X)
```

```
# calcualte "train error"
mean((y - predict(fit_ridge_cv, X)) ^ 2)
```

```
## [1] 132355.6
```

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

```
##  [1] 451.4406 449.8449 449.0434 448.7750 448.4817 448.1611 447.8110
##  [8] 447.4287 447.0116 446.5567 446.0608 445.5207 444.9328 444.2934
## [15] 443.5985 442.8440 442.0256 441.1391 440.1797 439.1429 438.0239
## [22] 436.8182 435.5211 434.1282 432.6354 431.0388 429.3350 427.5214
## [29] 425.5958 423.5570 421.4049 419.1405 416.7661 414.2853 411.7034
## [36] 409.0274 406.2656 403.4280 400.5263 397.5738 394.5849 391.5752
## [43] 388.5612 385.5596 382.5874 379.6614 376.7976 374.0109 371.3151
```

```
## [50] 368.7219 366.2406 363.8798 361.6453 359.5411 357.5687 355.7279
## [57] 354.0167 352.4318 350.9685 349.6214 348.3841 347.2501 346.2125
## [64] 345.2630 344.3984 343.6142 342.8974 342.2449 341.6503 341.1129
## [71] 340.6259 340.1803 339.7785 339.4154 339.0818 338.7807 338.5078
## [78] 338.2568 338.0289 337.8194 337.6265 337.4496 337.2849 337.1283
## [85] 336.9891 336.8463 336.7251 336.5932 336.4864 336.3592 336.2635
## [92] 336.1439 336.0558 335.9421 335.8632 335.7561 335.6854 335.5865
## [99] 335.5278
```

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

```
## [1] 335.5278
```

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

```
## [1] 368.7219
```

24.2 Lasso

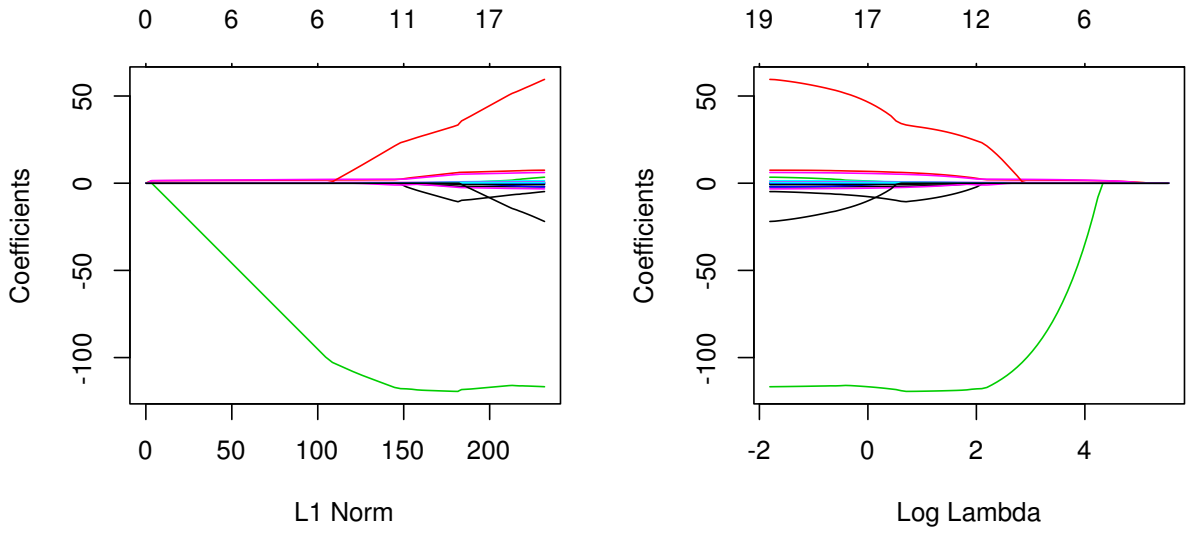
We now illustrate **lasso**, which can be fit using `glmnet()` with `alpha = 1` and seeks to minimize

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

Like ridge, lasso is not scale invariant.

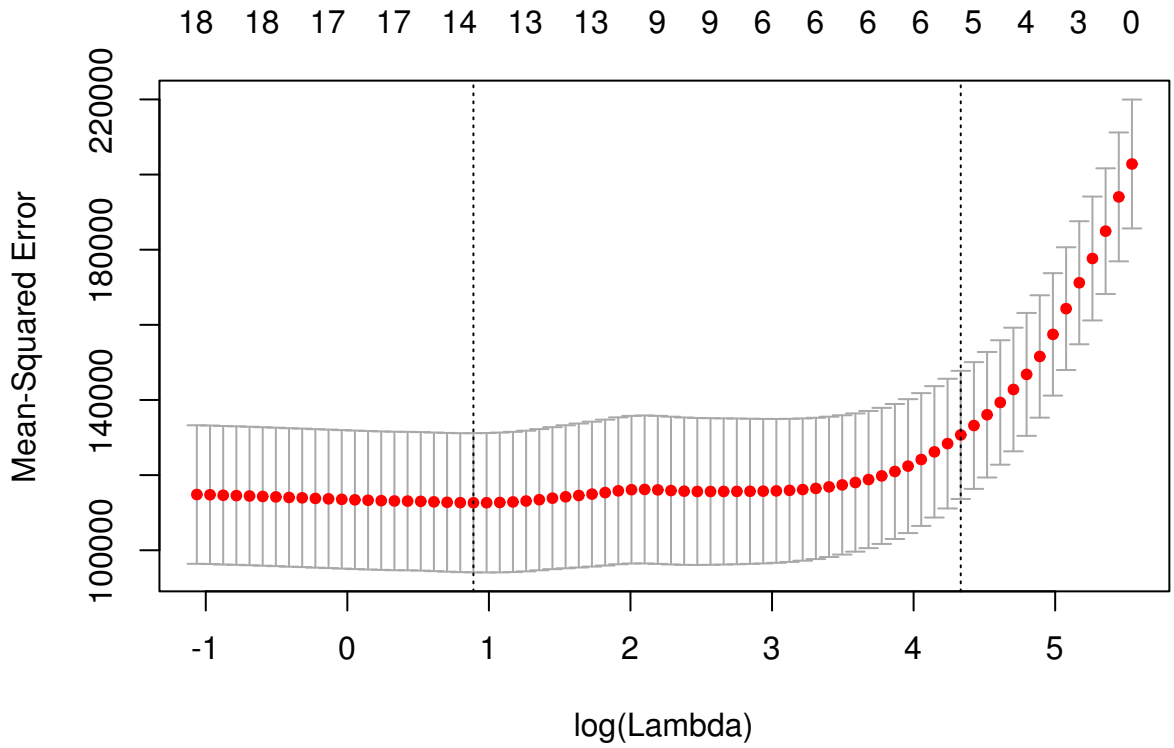
The two plots illustrate how much the coefficients are penalized for different values of λ. Notice some of the coefficients are forced to be zero.

```
par(mfrow = c(1, 2))
fit_lasso = glmnet(X, y, alpha = 1)
plot(fit_lasso)
plot(fit_lasso, xvar = "lambda", label = TRUE)
```



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.

```
# fitted coefficients, using 1-SE rule lambda, default behavior
coef(fit_lasso_cv)
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept) 144.37970485
## 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    .
```

```
# fitted coefficients, using minimum lambda
coef(fit_lasso_cv, s = "lambda.min")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept)  129.4155569
## AtBat         -1.6130155
## Hits           5.8058915
## HmRun          .
## Runs           .
## RBI            .
## Walks          4.8469340
## Years         -9.9724045
## CAtBat         .
## CHits          .
## CHmRun         0.5374550
## CRuns          0.6811938
## CRBI           0.3903563
## CWalks        -0.5560143
## LeagueN       32.4646094
## DivisionW   -119.3480842
## PutOuts        0.2741895
## Assists        0.1855978
## Errors        -2.1650837
## NewLeagueN     .
```

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

```
## [1] 15463.18
```

```
# fitted coefficients, using 1-SE rule lambda
coef(fit_lasso_cv, s = "lambda.1se")
```

```
## 20 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept) 144.37970485
## 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    .
```

```
# penalty term using 1-SE rule lambda
sum(coef(fit_lasso_cv, s = "lambda.1se")[-1] ^ 2)
```

```
## [1] 4.237431
```

```
# predict using minimum lambda
predict(fit_lasso_cv, X, s = "lambda.min")
```

```
# predict using 1-SE rule lambda, default behavior
predict(fit_lasso_cv, X)
```

```
# calcualte "train error"
mean((y - predict(fit_lasso_cv, X)) ^ 2)
```

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

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

```
##  [1] 450.3495 440.5207 430.0377 421.4959 413.7665 405.3558 396.8113
##  [8] 389.3391 383.1562 377.8777 373.2779 368.8759 364.9719 361.5263
## [15] 358.3109 355.2382 352.3493 349.8553 347.8022 346.1069 344.7039
## [22] 343.5492 342.6247 341.8690 341.2559 340.8130 340.4982 340.2962
## [29] 340.1888 340.1361 340.1018 340.0776 340.0599 340.0655 340.1805
## [36] 340.4240 340.6927 340.8601 340.7631 340.3277 339.6312 339.0318
## [43] 338.4873 337.9810 337.4379 336.8348 336.3078 335.9430 335.7339
## [50] 335.6337 335.6221 335.6618 335.8090 335.9819 336.1806 336.2858
## [57] 336.3268 336.4547 336.6192 336.7850 336.9737 337.1738 337.3437
## [64] 337.5691 337.7189 337.9343 338.1107 338.2982 338.4540 338.5786
## [71] 338.7744 338.8668
```

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

```
## [1] 335.6221
```

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

```
## [1] 361.5263
```

24.3 broom

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

```
library(broom)
# the output from the commented line would be immense
# fit_lasso_cv
tidy(fit_lasso_cv)
```

```
## # A tibble: 72 x 6
##    lambda estimate std.error conf.low conf.high nzero
##     <dbl>    <dbl>     <dbl>    <dbl>     <dbl> <int>
##  1   255.  202815.    17152.  185663.   219967.     0
##  2   233.  194058.    17151.  176908.   211209.     1
##  3   212.  184932.    16732.  168200.   201665.     2
##  4   193.  177659.    16486.  161173.   194145.     2
##  5   176.  171203.    16376.  154826.   187579.     3
##  6   160.  164313.    16333.  147980.   180646.     4
##  7   146.  157459.    16291.  141168.   173751.     4
##  8   133.  151585.    16289.  135296.   167874.     4
##  9   121.  146809.    16357.  130452.   163166.     4
## 10   111.  142792.    16455.  126337.   159246.     4
## # ... with 62 more rows
```

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

```
## # A tibble: 1 x 2
##   lambda.min lambda.1se
##        <dbl>      <dbl>
## 1       2.44       76.2
```

24.4 Simulated Data, $p > n$

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 $p > n$.

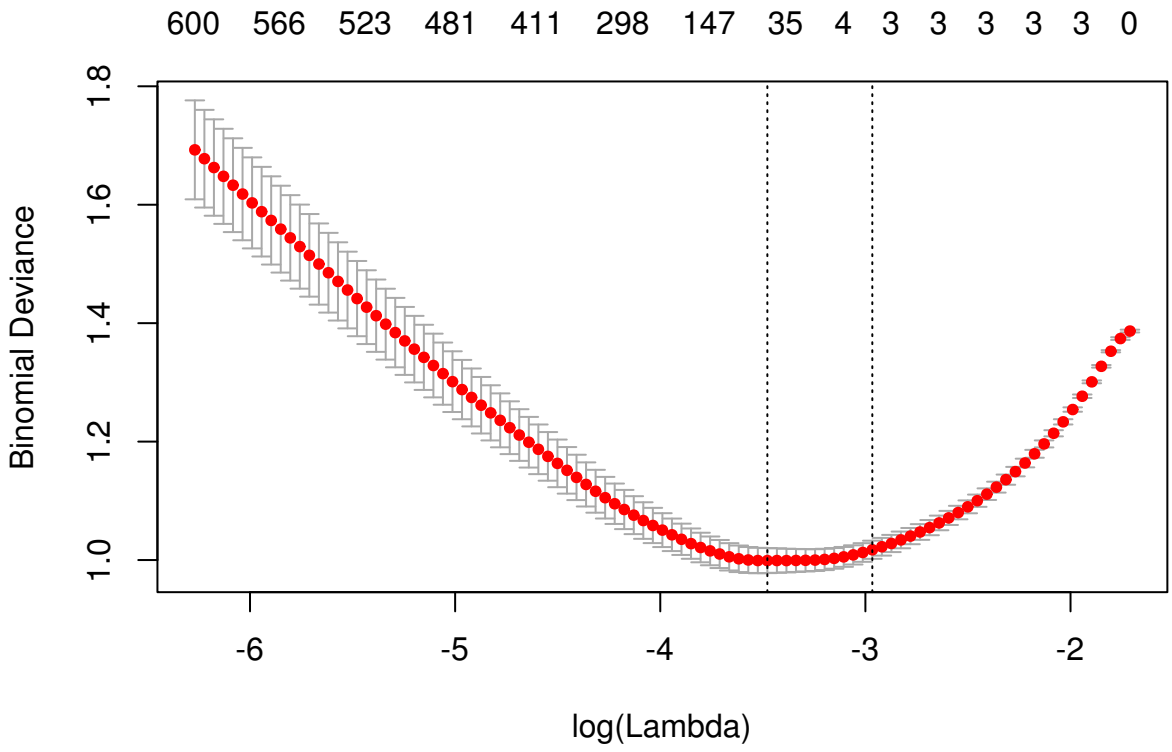
```
# glm(y ~ X, family = "binomial")
# will not converge
```

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
```

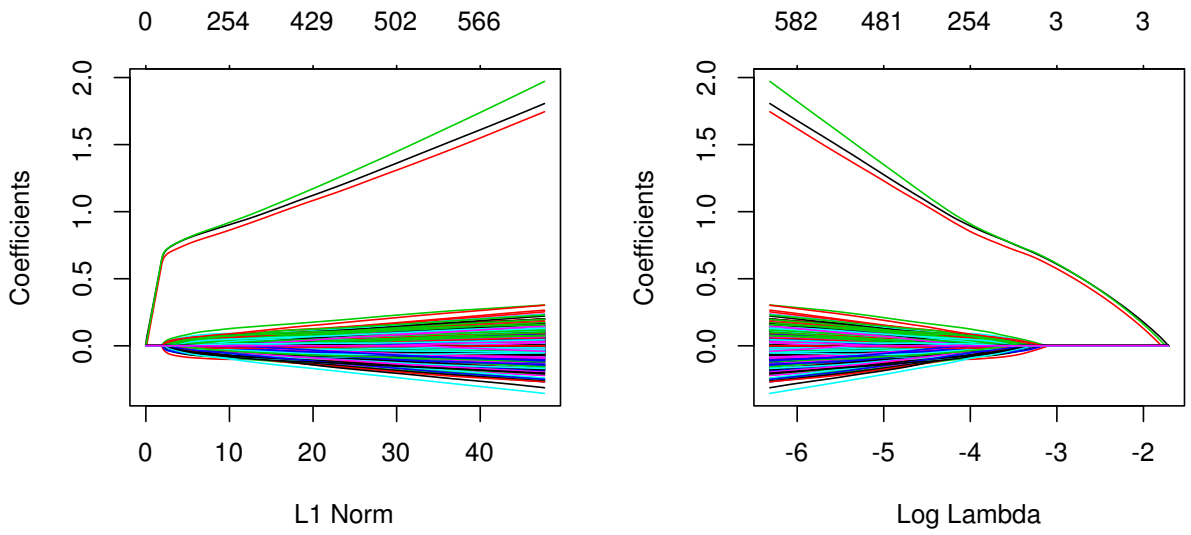
Notice, only the first three predictors generated are truly significant, and that is exactly what the suggested model finds.

```
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.

```
par(mfrow = c(1, 2))
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
```

The interaction between the `glmnet` and `caret` packages is sometimes frustrating, but for obtaining results for particular values of λ, we see it can be easily used. More on this next chapter.

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 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

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.

```
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 <fct>, Division <fct>, PutOuts <int>, Assists <int>,
## #   Errors <int>, Salary <dbl>, NewLeague <fct>
```

```
dim(Hitters)
```

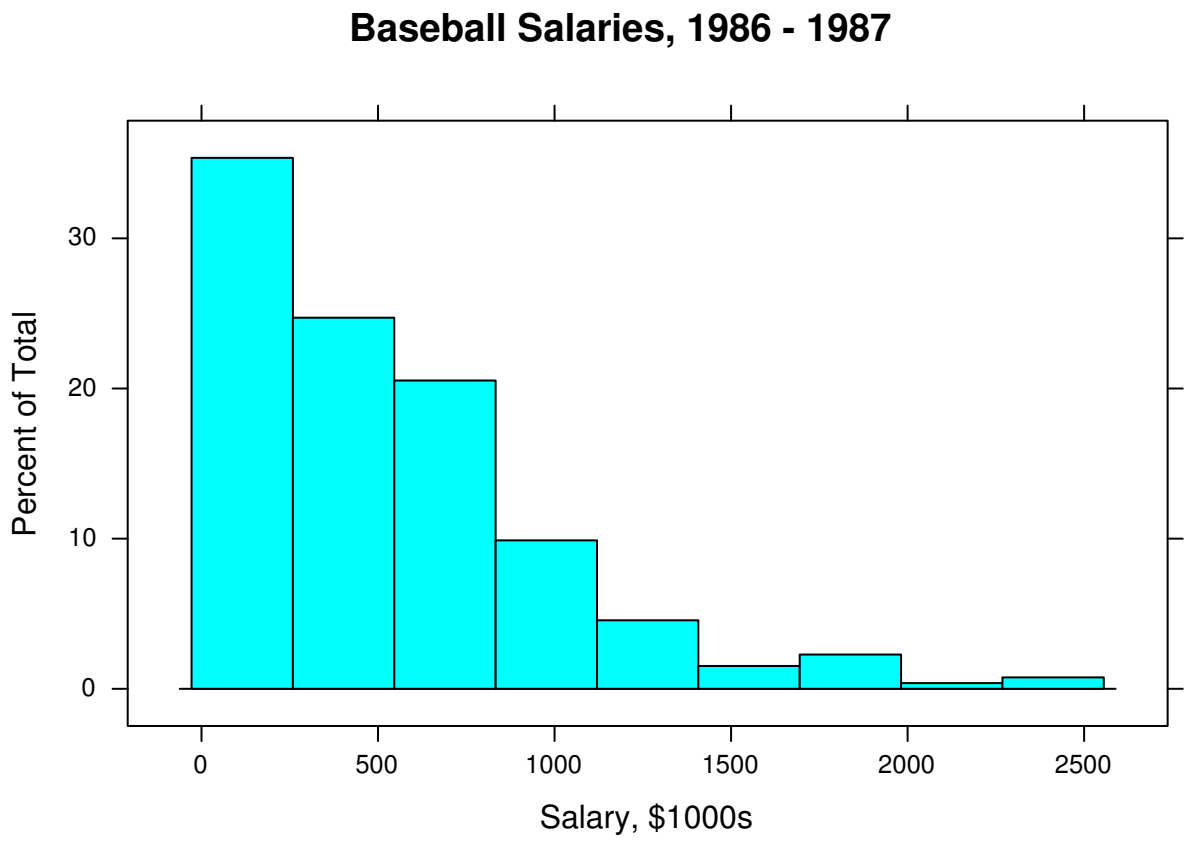
```
## [1] 263  20
```

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

```
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")
```



25.1 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(42)
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: 209, 211, 211, 212, 209
## Resampling results across tuning parameters:
##
##   alpha  lambda   RMSE   Rsquared  MAE
##   0.10    0.5106  327.7  0.4869    231.6
##   0.10    5.1056  327.4  0.4894    230.7
##   0.10   51.0564  334.3  0.4734    229.4
##   0.55    0.5106  327.6  0.4873    231.5
##   0.55    5.1056  328.1  0.4895    229.1
##   0.55   51.0564  338.7  0.4749    233.2
##   1.00    0.5106  327.6  0.4877    231.3
##   1.00    5.1056  331.3  0.4818    229.3
##   1.00   51.0564  348.8  0.4663    242.2
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.1 and lambda = 5.106.
```

Notice a few things with these results. First, we have tried three α values, `0.10`, `0.55`, and `1`. It is not entirely clear why `caret` doesn't use `0`. It likely uses `0.10` to fit a model close to ridge, but with some potential for sparsity.

Here, the best result uses $\alpha = 0.10$, so this result is somewhere between ridge and lasso, but closer to ridge.

```
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 = which(rownames(caret_fit$results) == rownames(caret_fit$bestTune))
  best_result = caret_fit$results[best, ]
  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  4.135 313.5     0.56 206.1  70.83     0.1254 24.37
```

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(42)
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] 305
```

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)
sum(coef(fit_lasso_cv) != 0)
```

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

```
sum(coef(fit_lasso_cv) == 0)
```

```
## [1] 186
```

25.2 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(42)
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, 6001, 6000, 6001
## Resampling results across tuning parameters:
##
##   alpha  lambda     Accuracy  Kappa
##   0.10   0.0001242  0.9731    0.4125
##   0.10   0.0012422  0.9731    0.3878
##   0.10   0.0124220  0.9679    0.0796
##   0.55   0.0001242  0.9731    0.4125
##   0.55   0.0012422  0.9731    0.3909
##   0.55   0.0124220  0.9684    0.1144
##   1.00   0.0001242  0.9731    0.4125
##   1.00   0.0012422  0.9732    0.4104
##   1.00   0.0124220  0.9693    0.1661
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were alpha = 1 and lambda = 0.001242.
```

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   0.1 0.001888   0.9732 0.3887   0.001843 0.07165
```

Here we see $\alpha = 0.1$, which is a mix, but close to ridge.

```
calc_acc = 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
calc_acc(actual = default_tst$default,
         predicted = predict(def_elnet_int, newdata = default_tst))
```

```
## [1] 0.9728
```

25.3 External Links

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

25.4 `rmarkdown`

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

Chapter 26

Trees

Chapter Status: This chapter was originally written using the `tree` packages. Currently being re-written to exclusively use the `rpart` package which seems more widely suggested and provides better plotting features.

```
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 ...
```

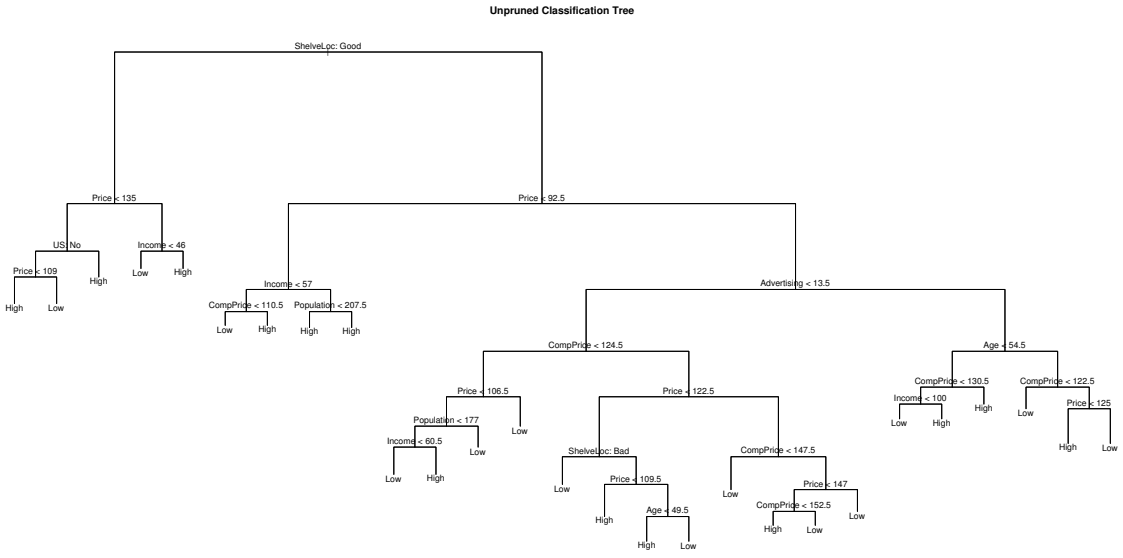
We first fit an unpruned classification tree using all of the predictors. Details of this process can be found using `?tree` and `?tree.control`

```
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
```

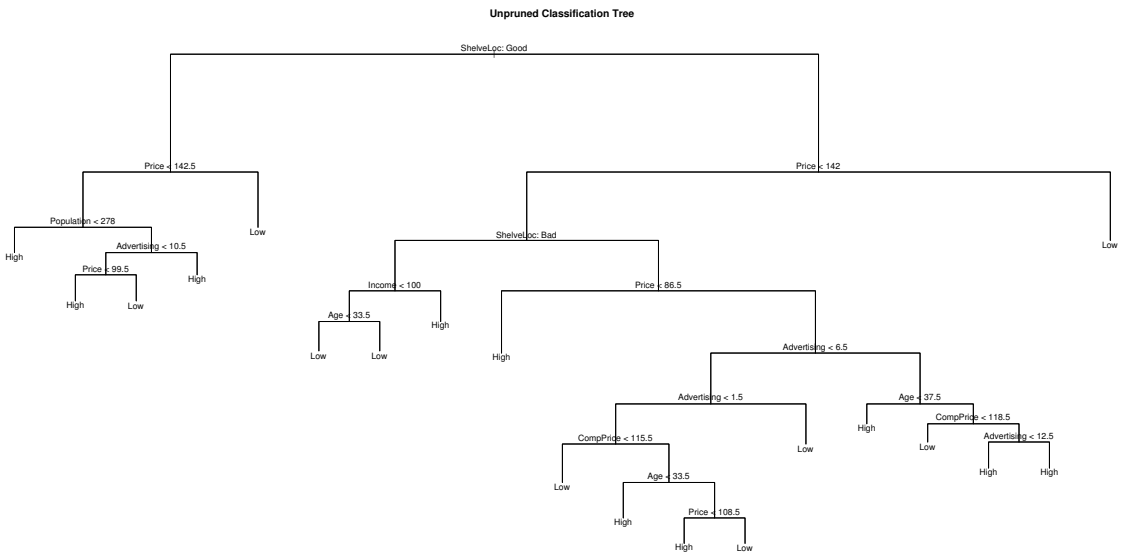
```
## [1] ShelveLoc   Price       Population  Advertising Income      Age
## [7] CompPrice
## 11 Levels: <leaf> CompPrice Income Advertising Population ... US
```

```
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
```

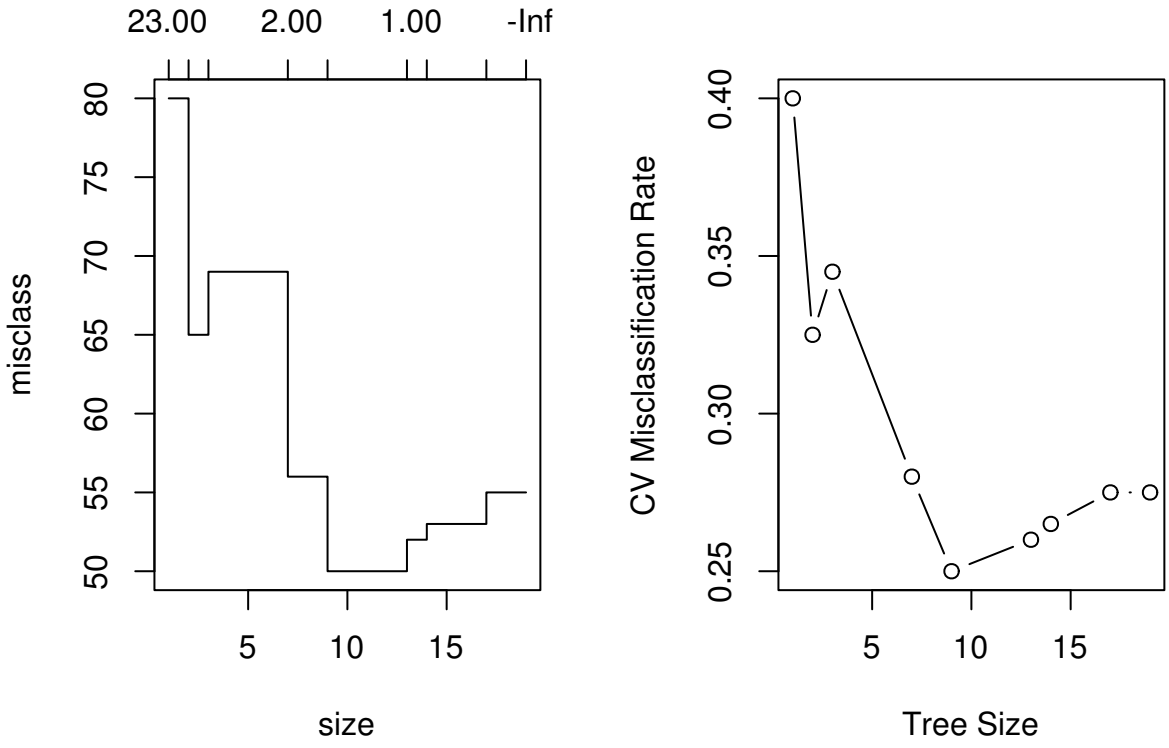
```
# number of terminal nodes in that tree
seat_tree_cv$size[min_idx]
```

```
## [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")
```



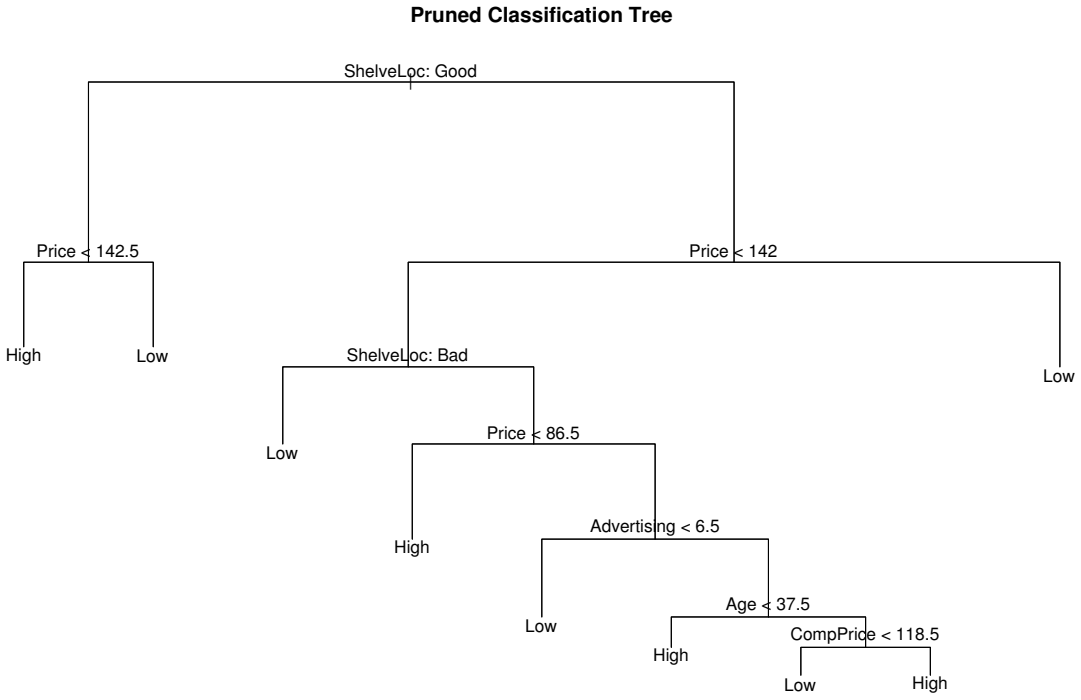
It appears that a tree of size 9 has the fewest misclassifications of the considered trees, via cross-validation.

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.

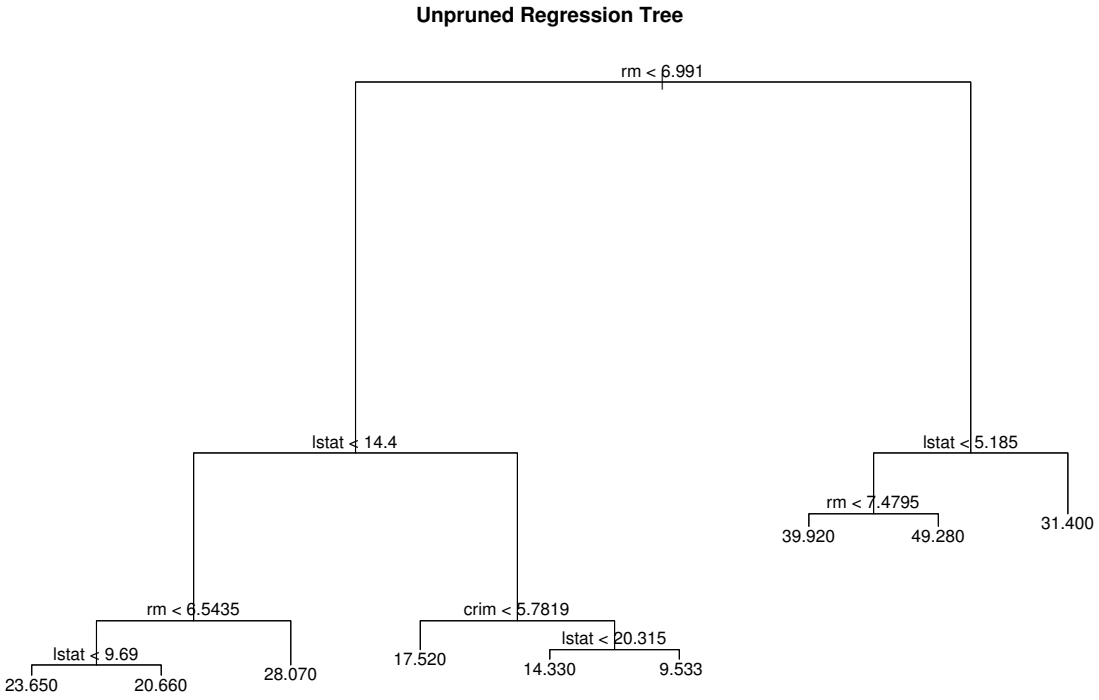
```
library(MASS)
set.seed(18)
boston_idx = sample(1:nrow(Boston), nrow(Boston) / 2)
boston_trn = Boston[boston_idx,]
boston_tst = Boston[-boston_idx,]
```

Then fit an unpruned regression tree to the training data.

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

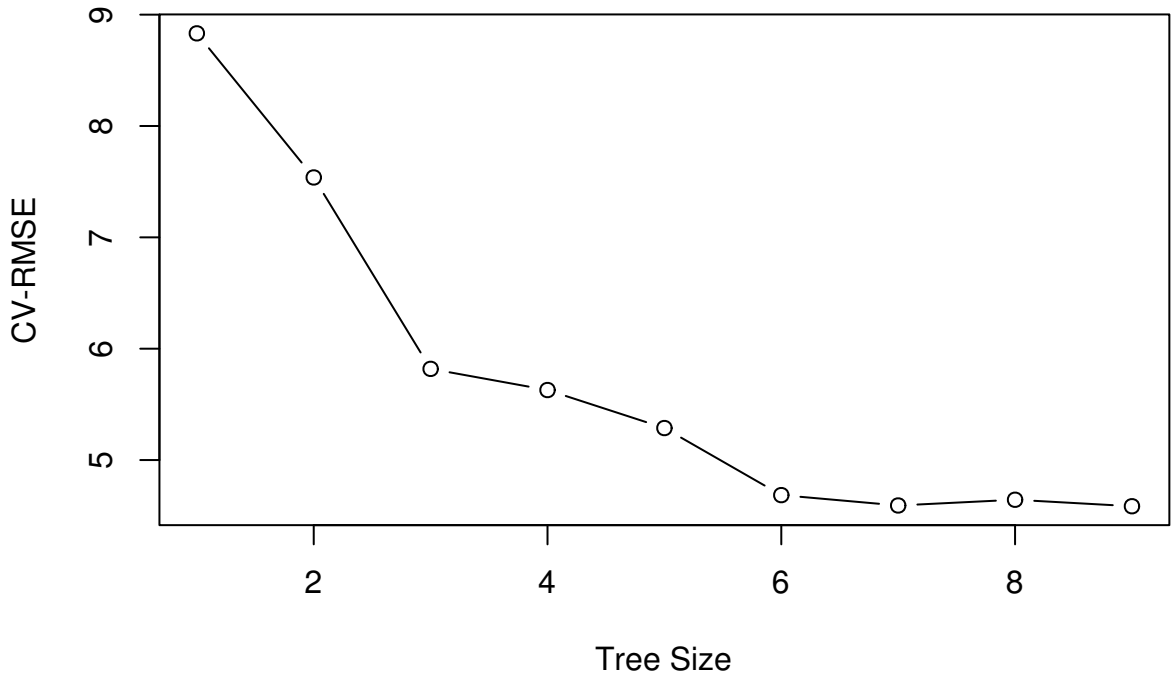
```
##
## Regression tree:
## tree(formula = medv ~ ., data = boston_trn)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"
## Number of terminal nodes:  9
## Residual mean deviance:  12.35 = 3013 / 244
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd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

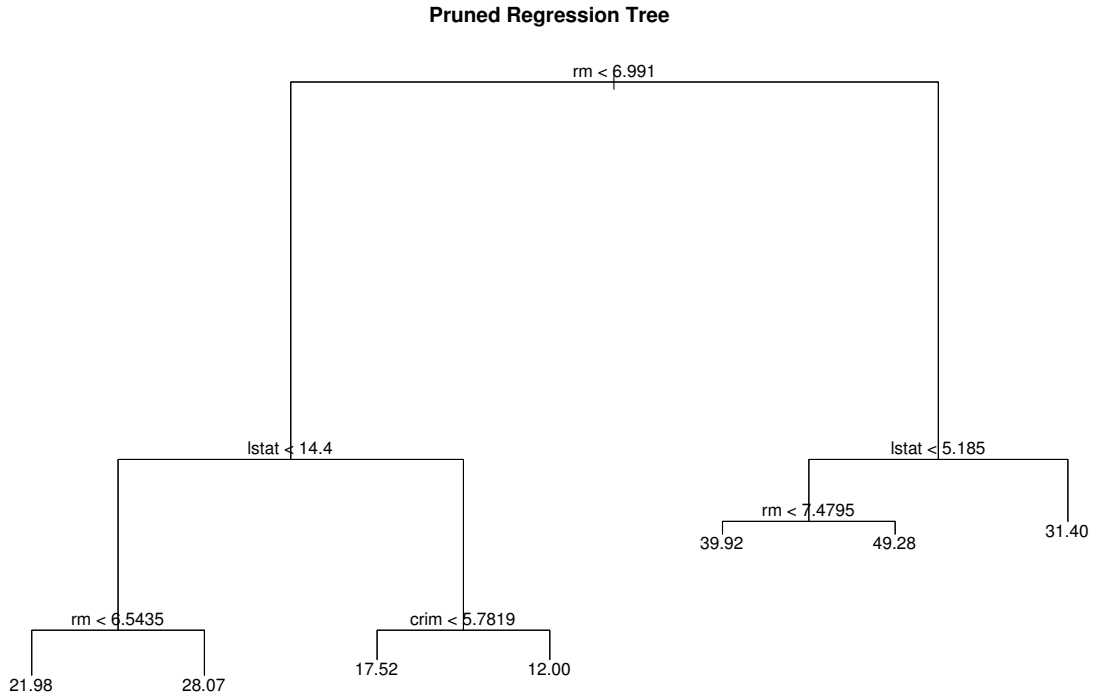


While the tree of size 9 does have the lowest RMSE, we'll prune to a size of 7 as it seems to perform just as well. (Otherwise we would not be pruning.) The pruned tree is, as expected, smaller and easier to interpret.

```
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
```

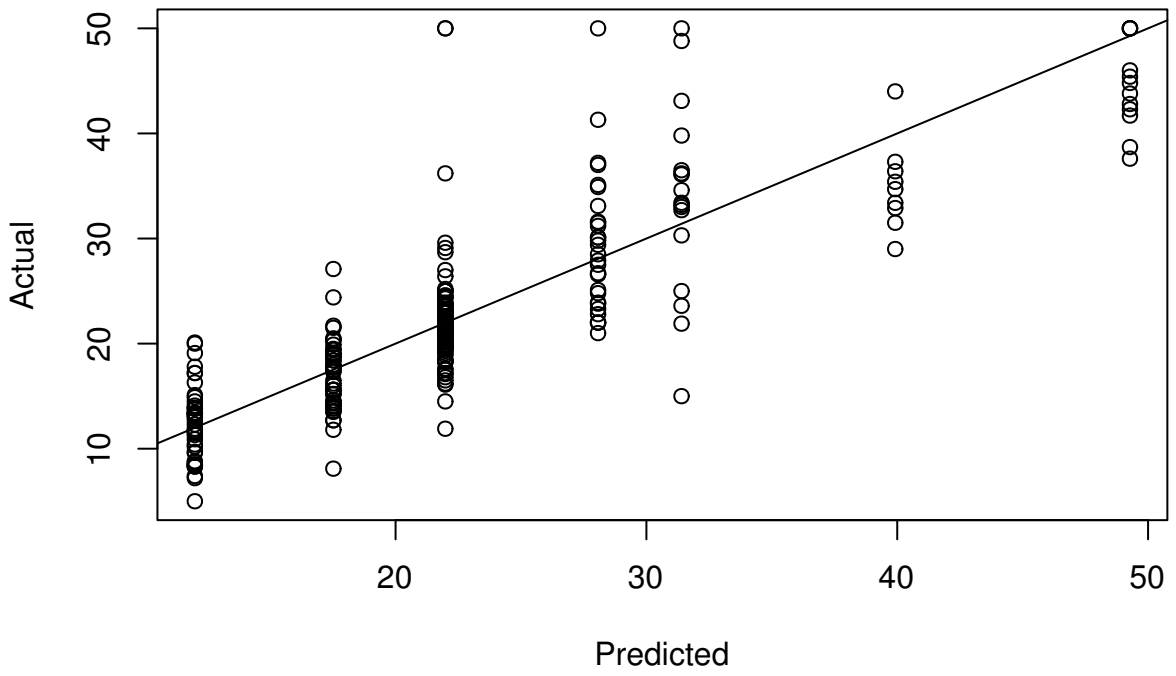
```
boston_prune_trn_pred = predict(boston_tree_prune, newdata = boston_trn)
rmse(boston_prune_trn_pred, boston_trn$medv)
```

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

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

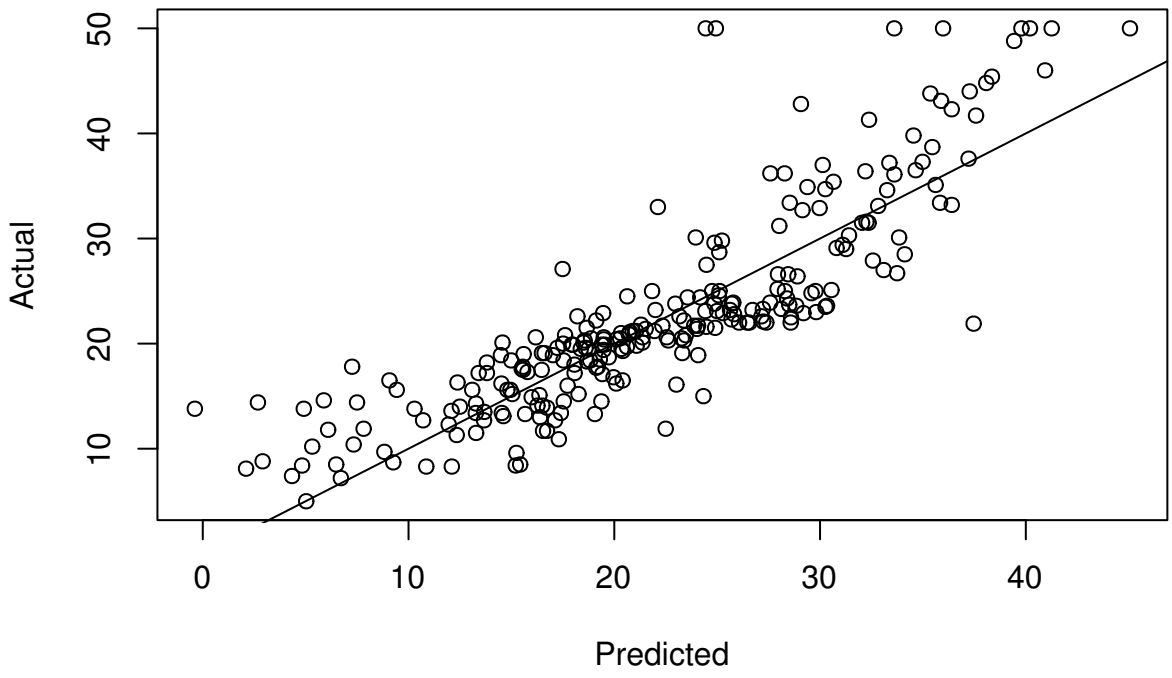
```
## [1] 5.331457
```

```
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)
```

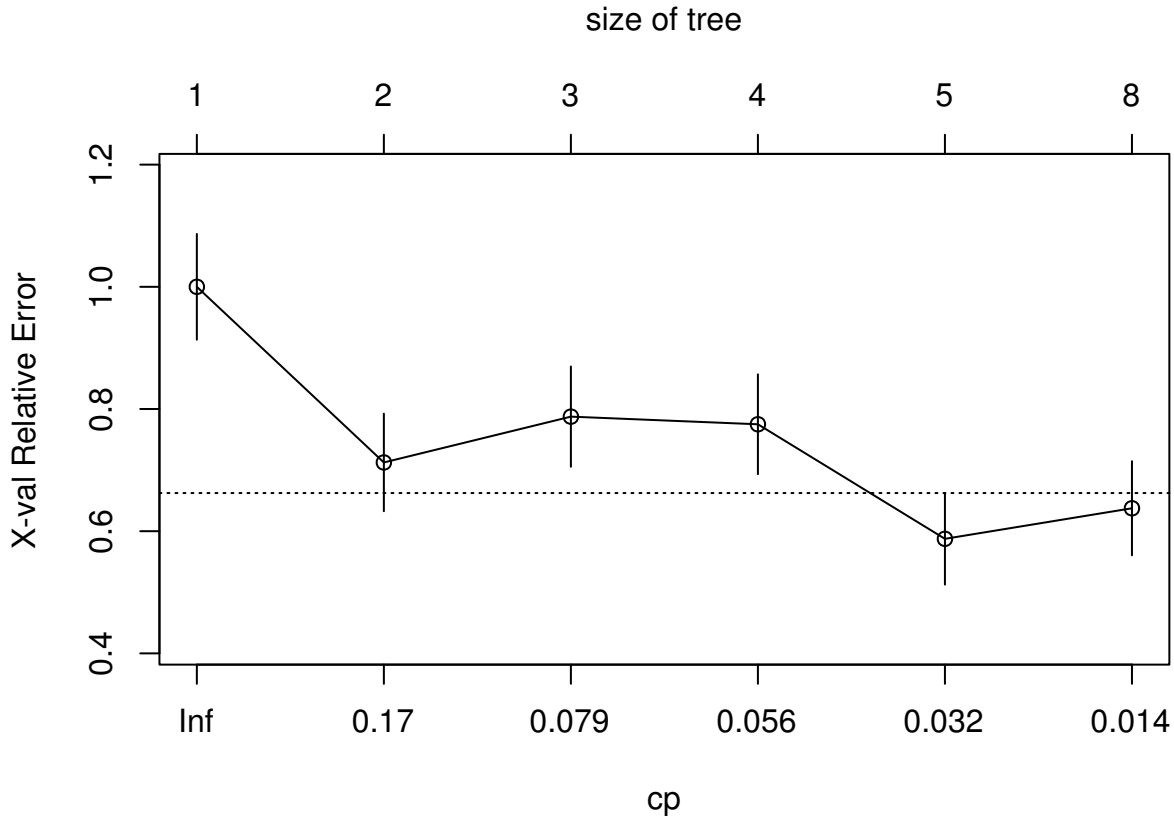
```
##   (Intercept)          crim            zn         indus          chas
##  43.340158284  -0.113490889   0.046881038   0.018046856   3.557944155
##           nox            rm           age           dis           rad
## -21.904534125   3.486780787  -0.010592511  -1.766227892   0.354167931
##           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
```

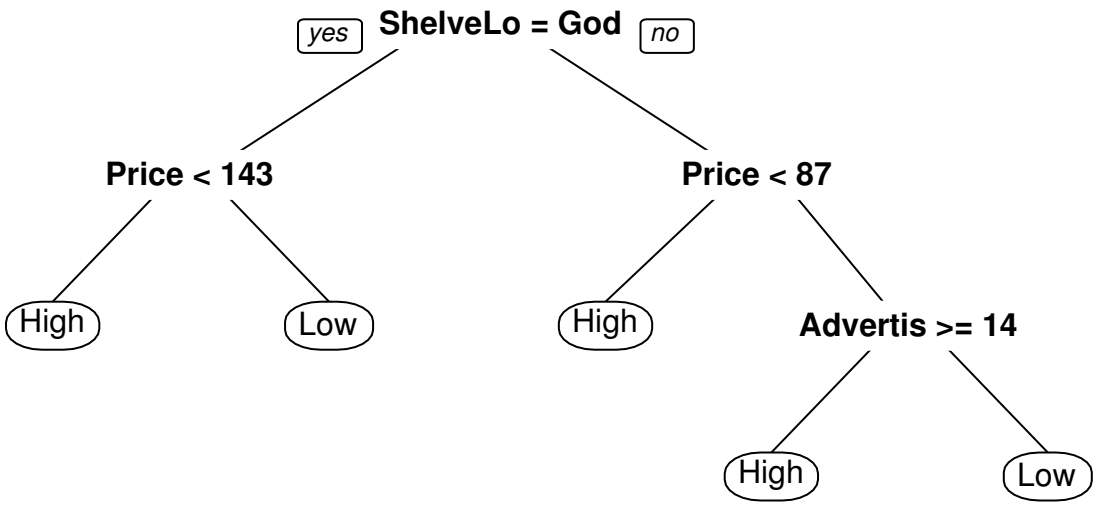
```
# Note: when you fit a tree using rpart, the fitting routine automatically
# performs 10-fold CV and stores the errors for later use
# (such as for pruning the tree)

# 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)
```



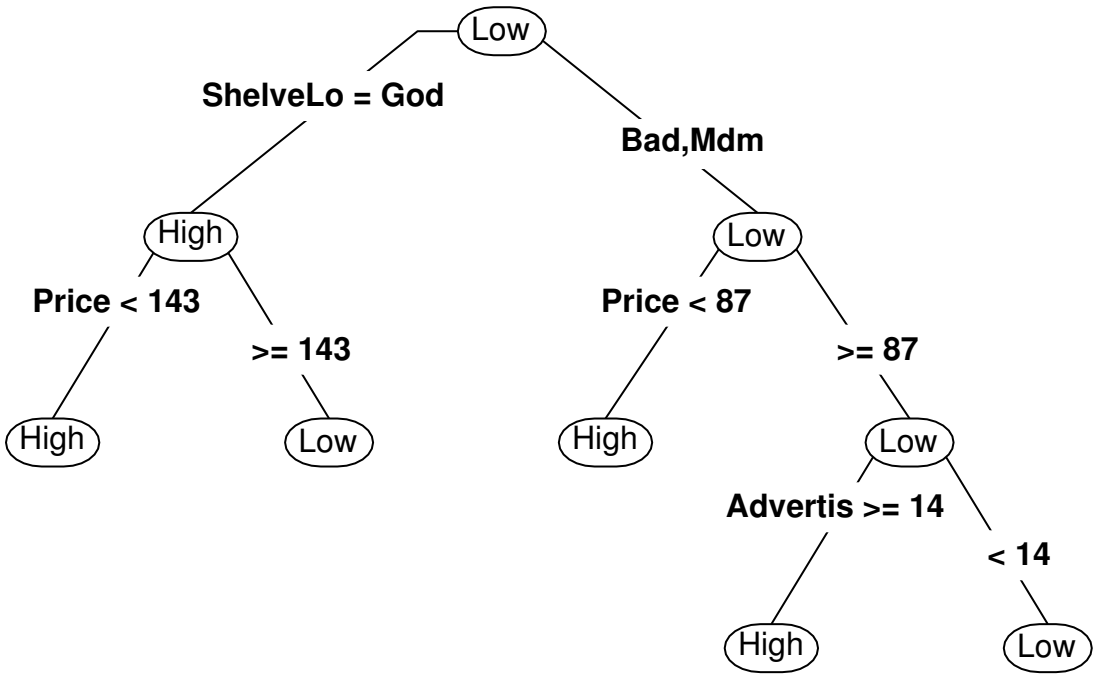

```
# find best value of cp
min_cp = seat_rpart$cptable[which.min(seat_rpart$cptable[,"xerror"]),"CP"]
min_cp
```

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

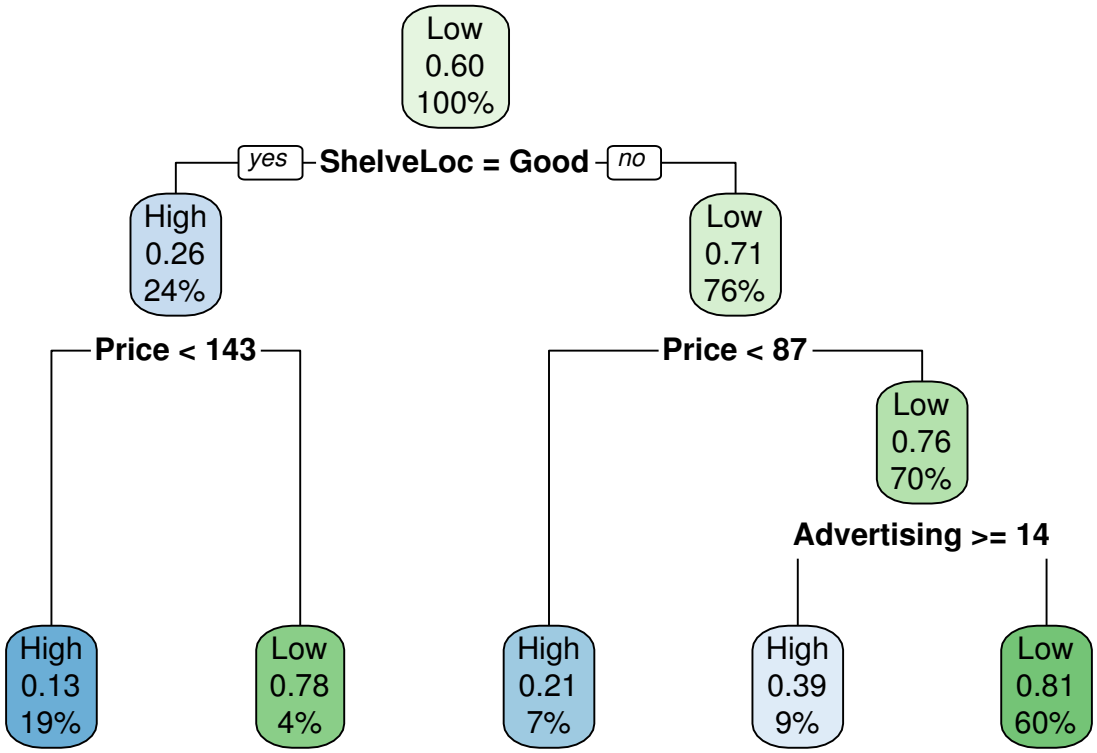
```
# prunce tree using best cp
seat_rpart_prune = prune(seat_rpart, cp = min_cp)

# 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 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

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

Chapter 27

Ensemble Methods

Chapter Status: Currently chapter is rather lacking in narrative and gives no introduction to the theory of the methods. The `R` code is in a reasonable place, but is generally a little heavy on the output, and could use some better summary of results. Using `Boston` for regression seems OK, but would like a better dataset for classification.

In this chapter, we'll 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.

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

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

```
library(rpart)
library(rpart.plot)
library(randomForest)
library(gbm)
library(caret)
library(MASS)
library(ISLR)
```

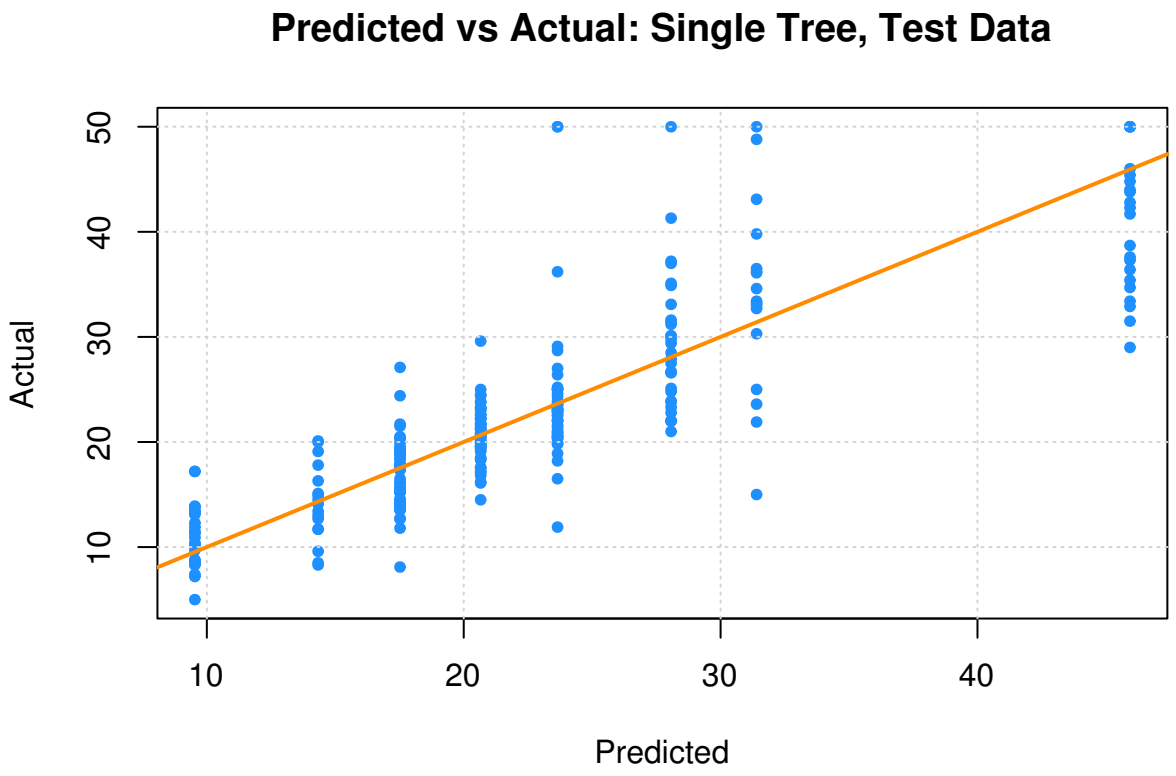
We first test-train split the data and fit a single tree using `rpart`.

```
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 = rpart(medv ~ ., data = boston_trn)
```

```
boston_tree_tst_pred = predict(boston_tree, newdata = boston_tst)
plot(boston_tree_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Single Tree, Test Data",
     col = "dodgerblue", pch = 20)
grid()
abline(0, 1, col = "darkorange", lwd = 2)
```




```
(tree_tst_rmse = calc_rmse(boston_tree_tst_pred, boston_tst$medv))
```

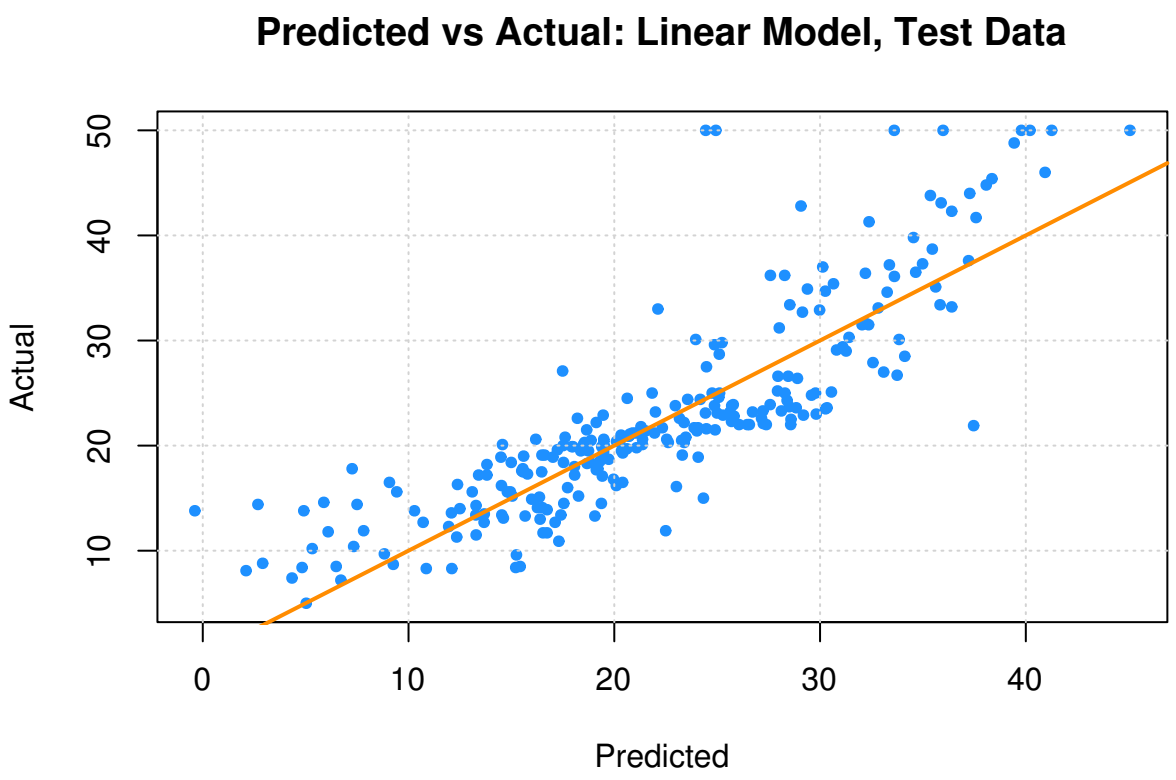
```
## [1] 5.458088
```

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.

```
boston_lm = lm(medv ~ ., data = boston_trn)
```

```
boston_lm_tst_pred = predict(boston_lm, newdata = boston_tst)
plot(boston_lm_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Linear Model, Test Data",
     col = "dodgerblue", pch = 20)
grid()
abline(0, 1, col = "darkorange", lwd = 2)
```




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

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

27.1.3 Bagging

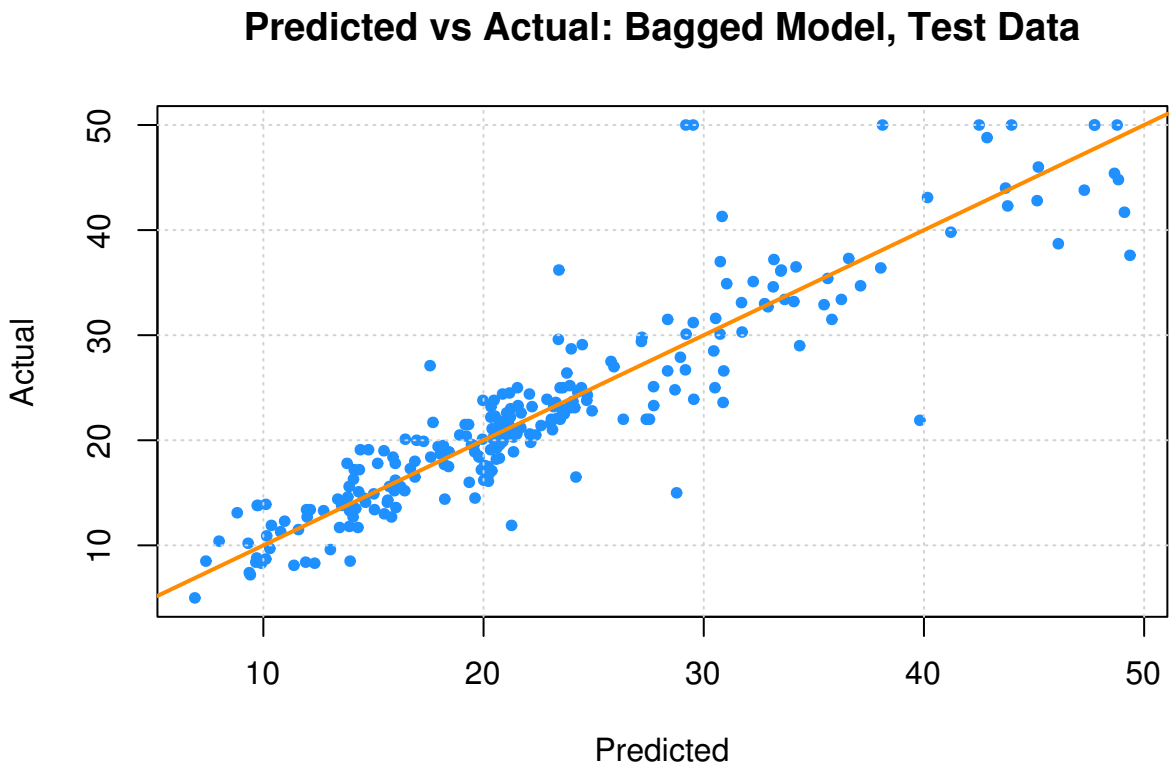
We now fit a bagged model, using the `randomForest` package. Bagging is actually a special case of a random forest where `mtry` is equal to p, the number of predictors.

```
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: 13.74459
##                     % Var explained: 81.53
```

```
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",
```

```
     col = "dodgerblue", pch = 20)
grid()
abline(0, 1, col = "darkorange", lwd = 2)
```

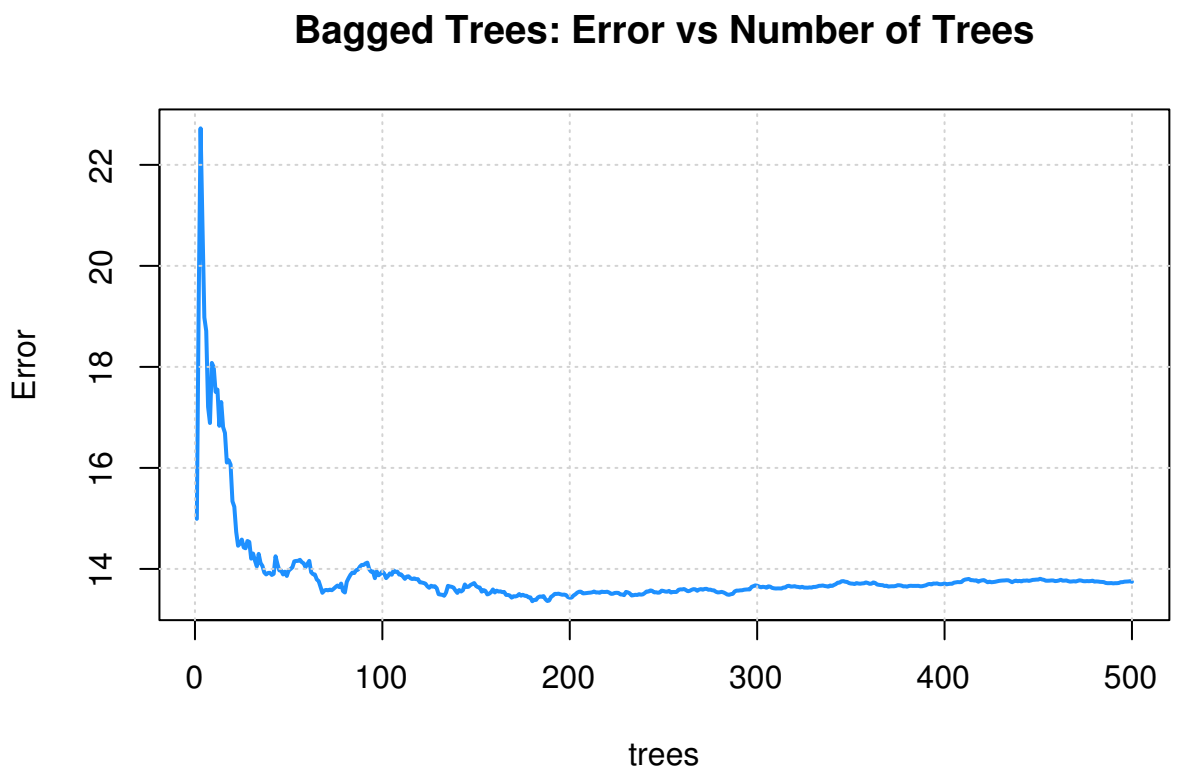



```
(bag_tst_rmse = calc_rmse(boston_bag_tst_pred, boston_tst$medv))
```

```
## [1] 3.843966
```

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, col = "dodgerblue", lwd = 2, main = "Bagged Trees: Error vs Number of Trees")
grid()
```



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
```

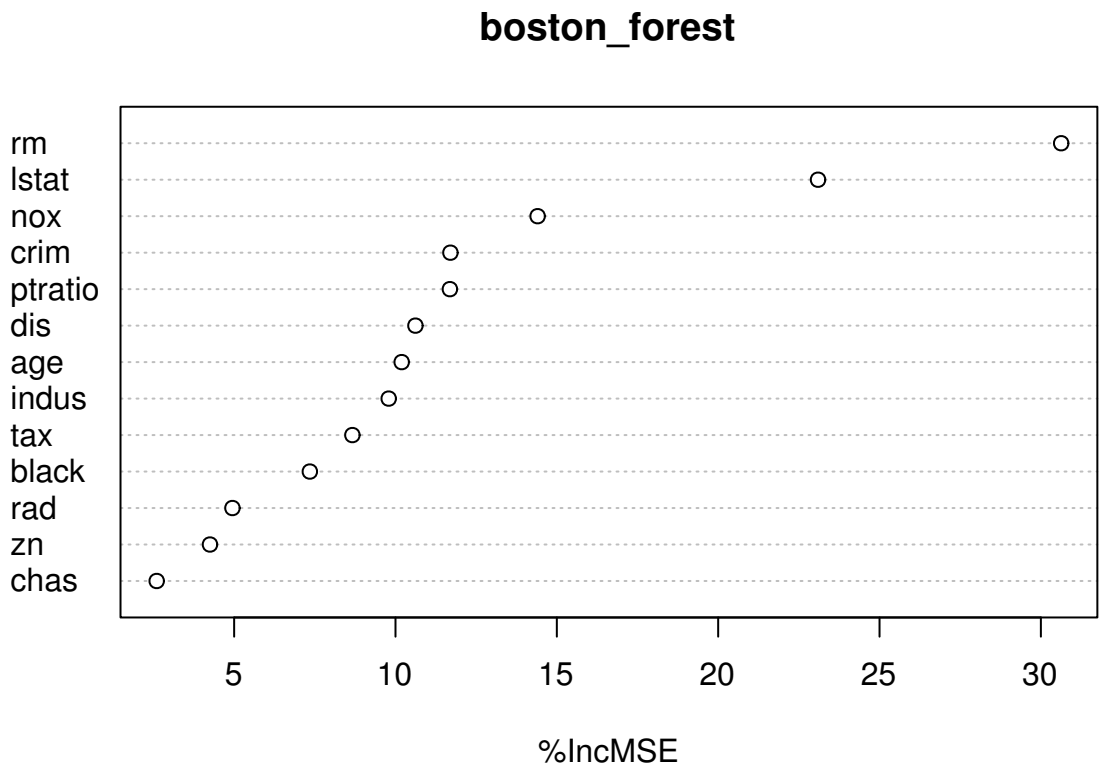
```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston_trn, mtry = 4,      importance = TRUE, ntrees = 500)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 12.78942
##                     % Var explained: 82.82
```

```
importance(boston_forest, type = 1)
```

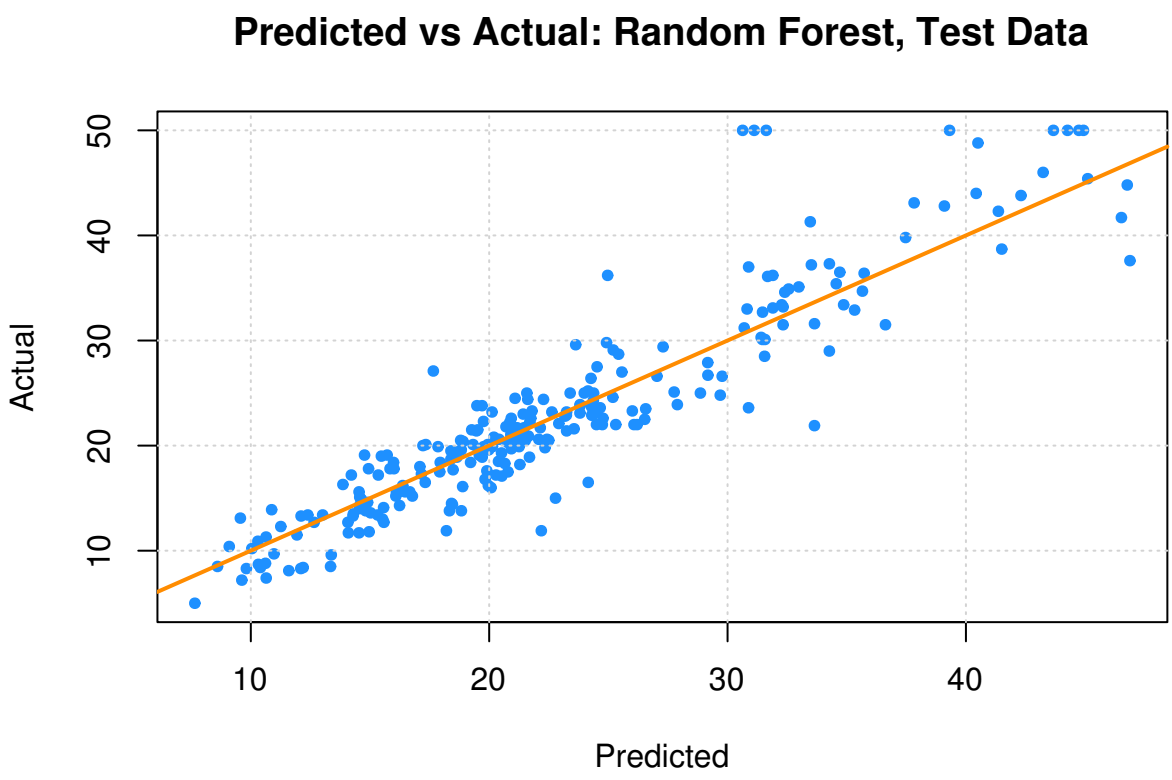
```
##           %IncMSE
## crim    11.702184
## zn       4.251232
## indus    9.789632
## chas     2.599685
## nox     14.404374
```

```
## rm      30.629328
## age     10.192164
## dis     10.617532
## rad      4.947145
## tax      8.663865
## ptratio 11.686099
## black    7.345671
## lstat   23.094159
```

```
varImpPlot(boston_forest, type = 1)
```




```
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",
     col = "dodgerblue", pch = 20)
grid()
abline(0, 1, col = "darkorange", lwd = 2)
```




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

```
## [1] 3.701805
```

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

Here we note three RMSEs. The training RMSE (which is optimistic), the OOB RMSE (which is a reasonable estimate of the test error) and the test RMSE. Also note that variables importance was calculated.

```
##       Data    Error
## 1 Training 1.558111
## 2      OOB 3.576229
## 3     Test 3.701805
```

27.1.5 Boosting

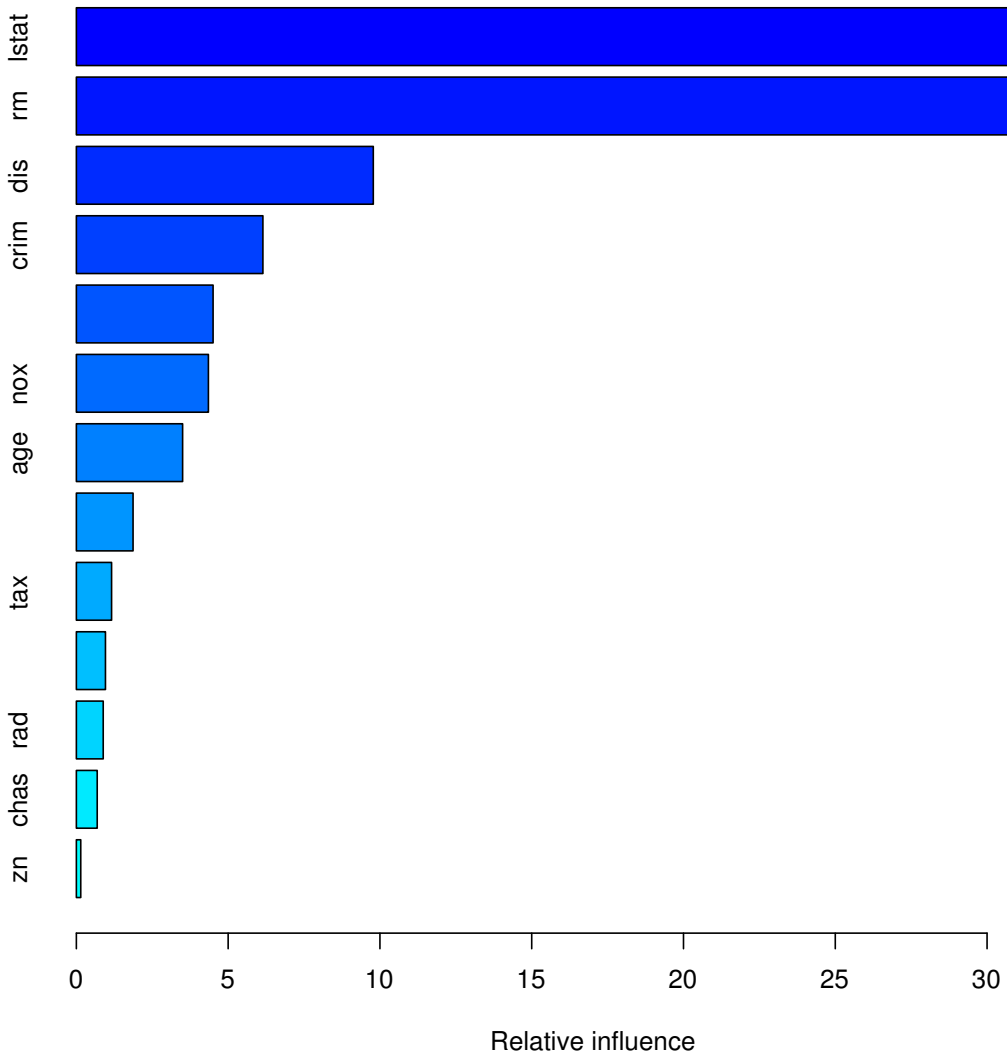
Lastly, we try a boosted model, which by default will produce a nice **variable importance** plot as well as plots of the marginal effects of the predictors. We use the `gbm` package.

```
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.
```

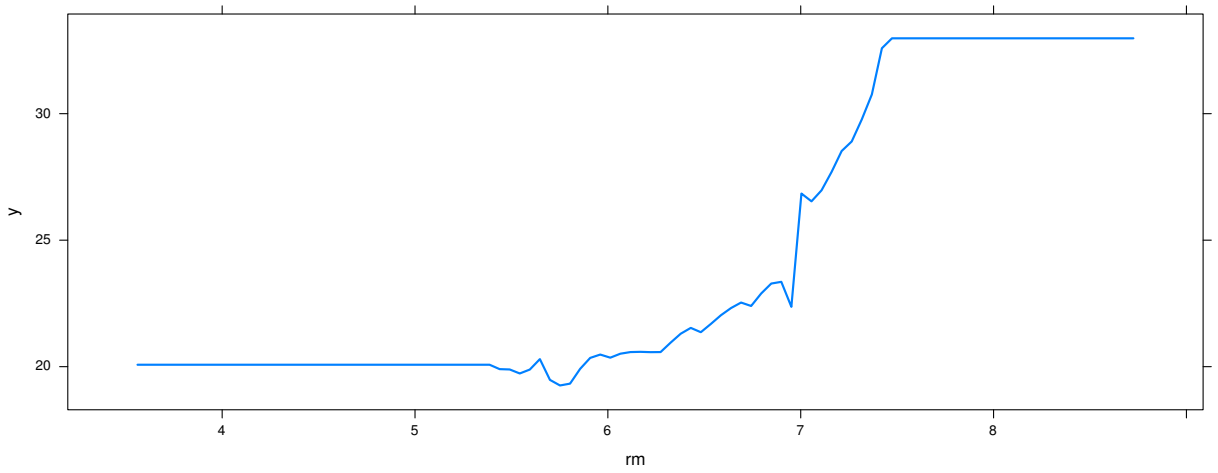
```
tibble::as_tibble(summary(booston_boost))
```



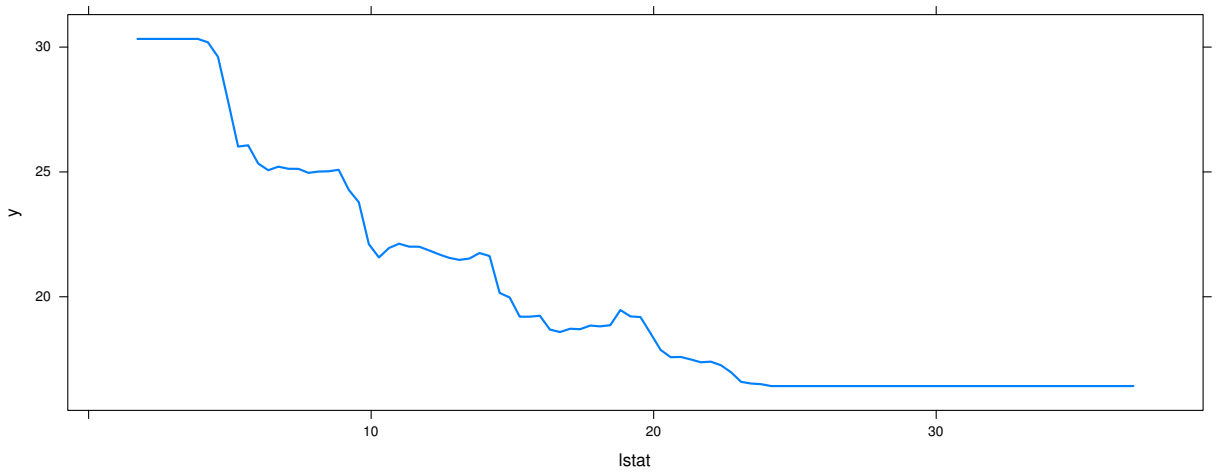

```
## # A tibble: 13 x 2
##    var     rel.inf
##  * <fct>     <dbl>
##  1 lstat    34.4
##  2 rm       31.6
##  3 dis       9.78
##  4 crim      6.15
##  5 black     4.50
##  6 nox       4.35
##  7 age       3.50
```

```
##  8 ptratio   1.87
##  9 tax       1.16
## 10 indus     0.959
## 11 rad       0.885
## 12 chas      0.687
## 13 zn        0.145
```

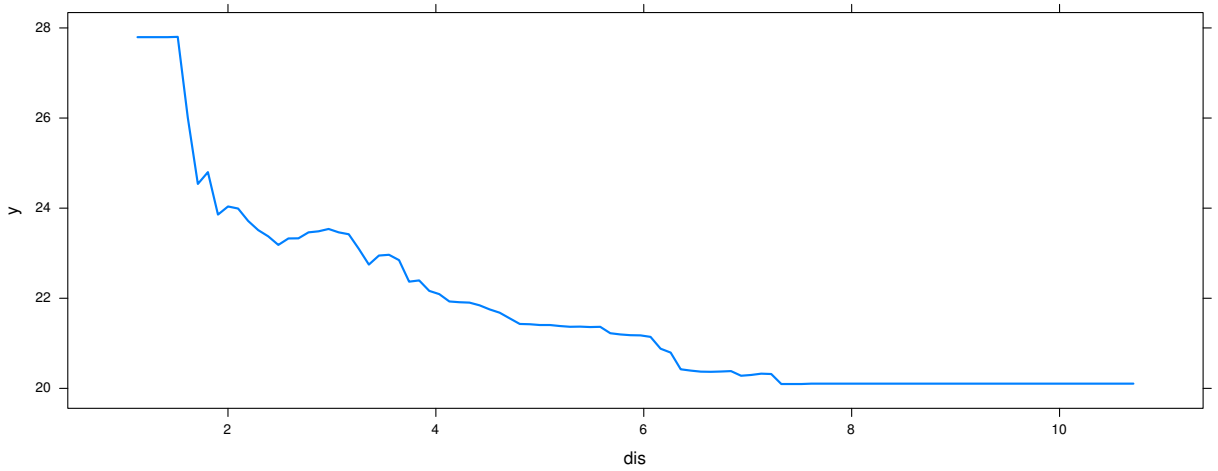
```
par(mfrow = c(1, 3))
plot(booston_boost, i = "rm", col = "dodgerblue", lwd = 2)
```




```
plot(booston_boost, i = "lstat", col = "dodgerblue", lwd = 2)
```



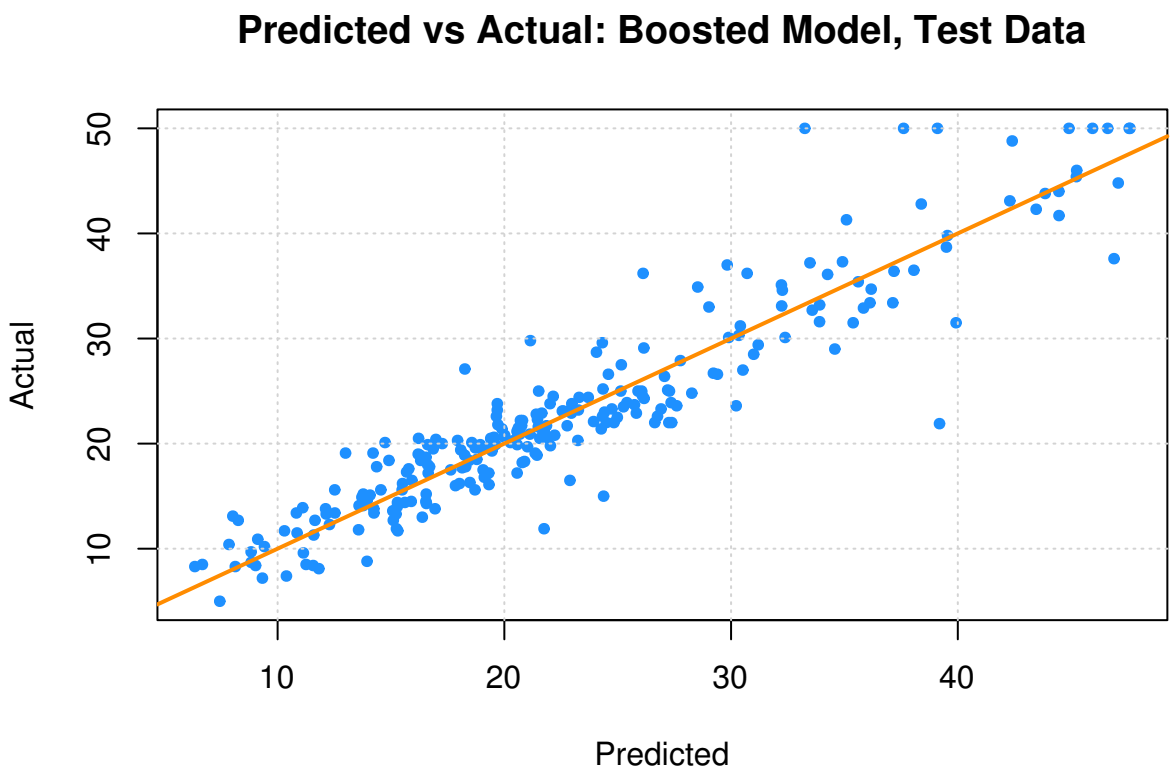

```
plot(booston_boost, i = "dis", col = "dodgerblue", lwd = 2)
```




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

```
## [1] 3.43382
```

```
plot(boston_boost_tst_pred, boston_tst$medv,
     xlab = "Predicted", ylab = "Actual",
     main = "Predicted vs Actual: Boosted Model, Test Data",
     col = "dodgerblue", pch = 20)
grid()
abline(0, 1, col = "darkorange", 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  5.458088
## 2  Linear Model  5.125877
## 3       Bagging  3.843966
## 4 Random Forest  3.701805
## 5      Boosting  3.433820
```

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:

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

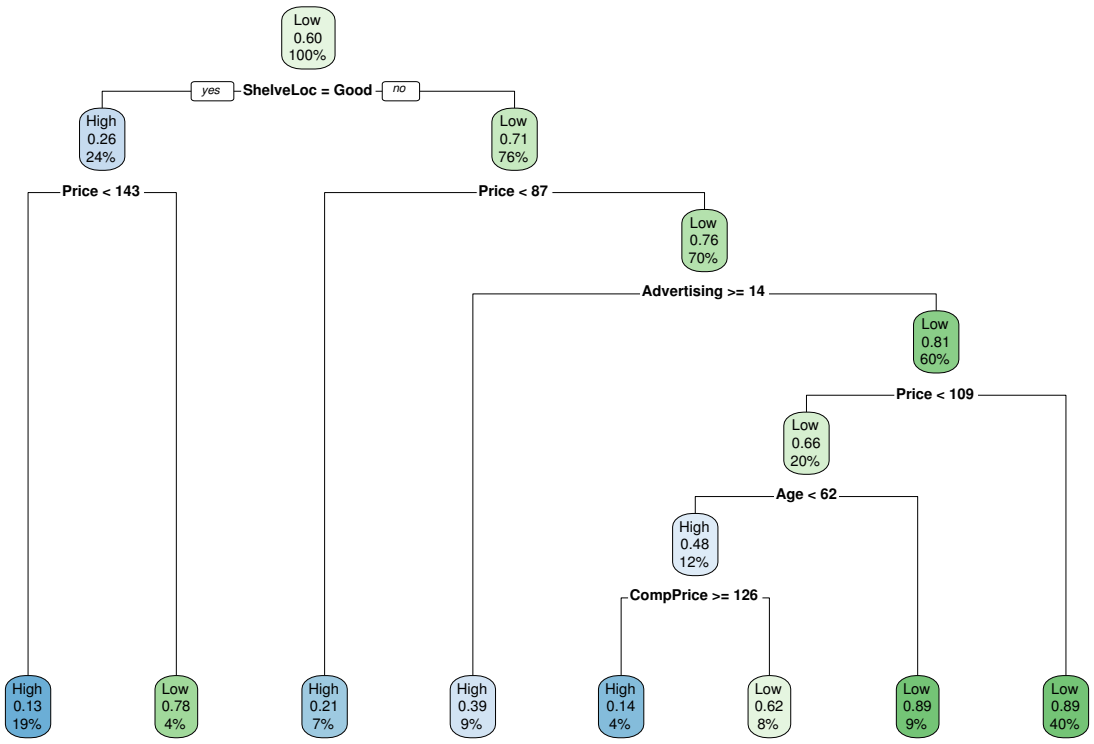
```
data(Carseats)
Carseats$Sales = as.factor(ifelse(Carseats$Sales <= 8, "Low", "High"))
```

```
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 = rpart(Sales ~ ., data = seat_trn)
```

```
rpart.plot(seat_tree)
```




```
seat_tree_tst_pred = predict(seat_tree, seat_tst, type = "class")
table(predicted = seat_tree_tst_pred, actual = seat_tst$Sales)
```

```
##          actual
## predicted High Low
##      High   64  26
##      Low    20  90
```

```
(tree_tst_acc = calc_acc(predicted = seat_tree_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 = calc_acc(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%
## Confusion matrix:
##      High Low class.error
## High   51  29   0.3625000
## Low    13 107   0.1083333
```

```
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   66  19
##      Low    18  97
```

```
(bag_tst_acc = calc_acc(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   50  30   0.3750000
## Low    14 106   0.1166667
```

```
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   64  18
##      Low    20  98
```

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

```
## [1] 0.81
```

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  16
```

```
##      Low    14 100
```

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

```
## [1] 0.85
```

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        0.810
## 5            Boosting        0.850
```

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 require much 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    0.740     0.4090909
##    2    0.790     0.5434783
##    3    0.770     0.5043103
##    4    0.785     0.5376344
##    5    0.790     0.5493562
##    6    0.785     0.5415778
##    7    0.790     0.5493562
##    8    0.805     0.5824411
##    9    0.785     0.5376344
##   10    0.805     0.5824411
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 8.
```

```
calc_acc(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
## 8    8
```

Based on these results, we would select the random forest model with an `mtry` of 8. Note that based on the OOB estimates, the bagging model is expected to perform worse than this selected 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)
```

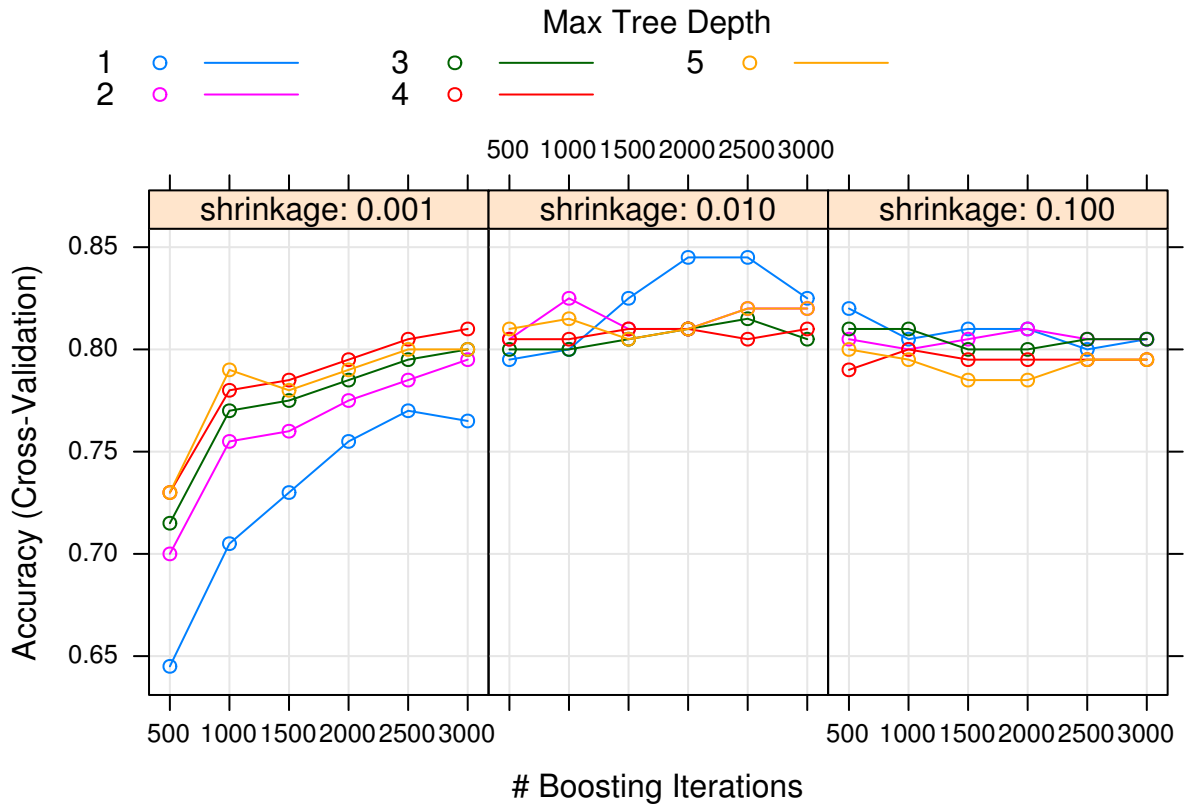
We now train the model using all possible combinations of the tuning parameters we just specified.

```
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 summarizing the results.

```
plot(seat_gbm_tune)
```




```
calc_acc(predict(seat_gbm_tune, seat_tst), seat_tst$Sales)
```

```
## [1] 0.85
```

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
```

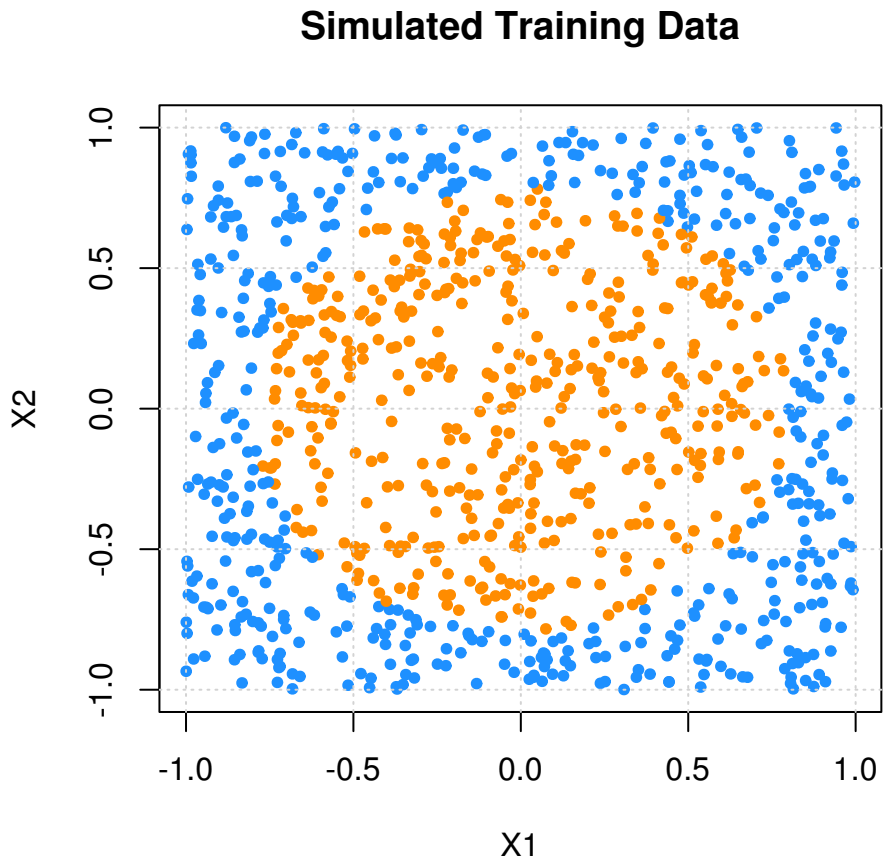
```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 34    2000                 1      0.01             10
```

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))
```

```
sim_trn_col = ifelse(sim_trn$class == 1, "darkorange", "dodgerblue")
plot(sim_trn$X1, sim_trn$X2, col = sim_trn_col,
```

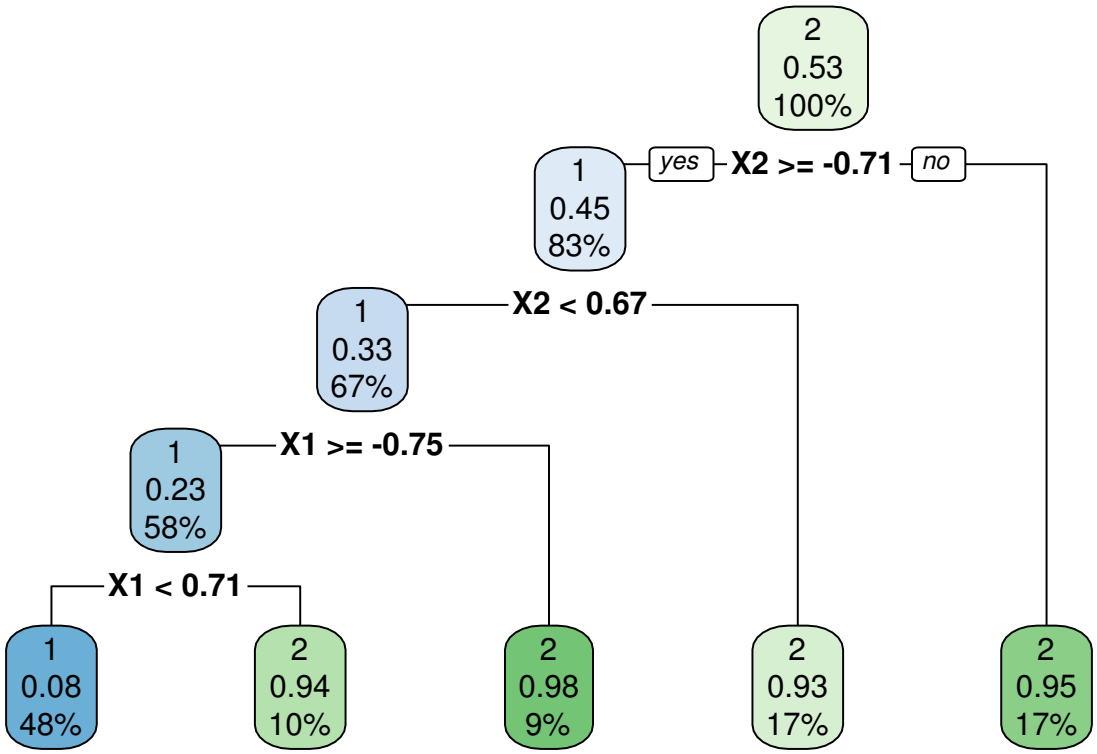
```
     xlab = "X1", ylab = "X2", main = "Simulated Training Data", pch = 20)
grid()
```




```
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)
rpart.plot(sim_tree_cv$finalModel)
```



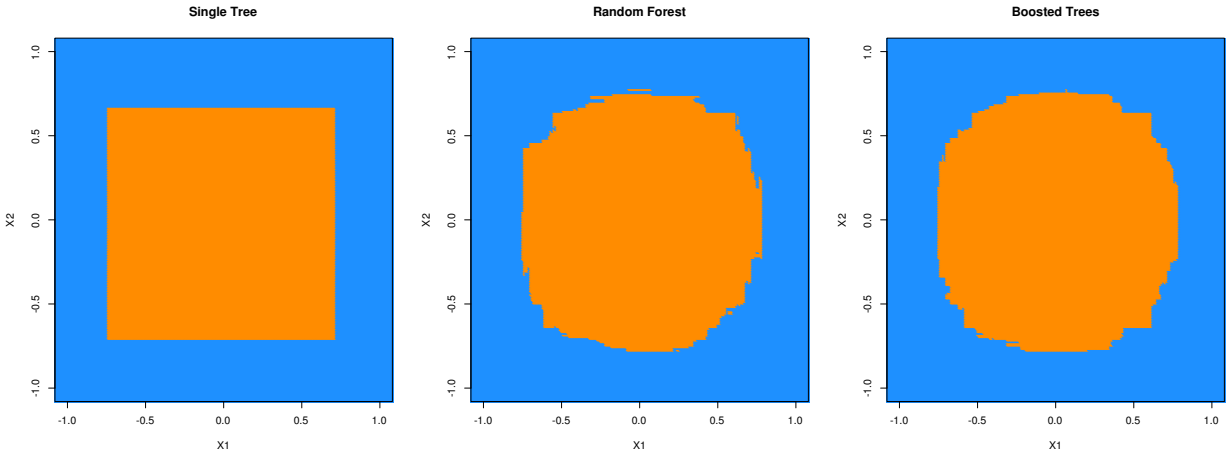

```
rf_grid = expand.grid(mtry = c(1, 2))
sim_rf_oob = train(class ~ .,
                   data = sim_trn,
                   trControl = oob,
                   tuneGrid = rf_grid)
```

```
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$X1) - 1, max(sim_tst$X1) + 1, by = 0.01),
  X2 = seq(min(sim_tst$X2) - 1, max(sim_tst$X2) + 1, 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)
```

```
tree_col = ifelse(tree_pred == 1, "darkorange", "dodgerblue")
rf_col   = ifelse(rf_pred == 1, "darkorange", "dodgerblue")
gbm_col  = ifelse(gbm_pred == 1, "darkorange", "dodgerblue")
```

```
par(mfrow = c(1, 3))
plot(plot_grid$X1, plot_grid$X2, col = tree_col,
     xlab = "X1", ylab = "X2", pch = 20, main = "Single Tree",
     xlim = c(-1, 1), ylim = c(-1, 1))
plot(plot_grid$X1, plot_grid$X2, col = rf_col,
     xlab = "X1", ylab = "X2", pch = 20, main = "Random Forest",
     xlim = c(-1, 1), ylim = c(-1, 1))
plot(plot_grid$X1, plot_grid$X2, col = gbm_col,
     xlab = "X1", ylab = "X2", pch = 20, main = "Boosted Trees",
     xlim = c(-1, 1), ylim = c(-1, 1))
```



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 3.5.1. The following packages (and their dependencies) were loaded when knitting this file:

```
##  [1] "mlbench"      "ISLR"         "MASS"         "caret"
##  [5] "ggplot2"      "lattice"      "gbm"          "randomForest"
##  [9] "rpart.plot"   "rpart"
```

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

We now use the `Sonar` dataset from the `mlbench` package to explore a new regularization method, **regularized discriminant analysis** (RDA), which combines the LDA and QDA. This is similar to how elastic net combines the ridge and lasso.

31.1 Sonar 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(mlbench)
library(caret)
library(glmnet)
library(klaR)
```

```
data(Sonar)
```

```
#View(Sonar)
```

```
table(Sonar$Class) / nrow(Sonar)
```

```
##
##         M         R
## 0.5336538 0.4663462
```

```
ncol(Sonar) - 1
```

```
## [1] 60
```

31.2 RDA

Regularized discriminant analysis uses the same general setup as LDA and QDA but estimates the covariance in a new way, which combines the covariance of QDA ($\hat{\Sigma}_k$) with the covariance of LDA ($\hat{\Sigma}$) using a tuning parameter λ.

$$\hat{\Sigma}_k(\lambda) = (1 - \lambda)\hat{\Sigma}_k + \lambda\hat{\Sigma}$$

Using the `rda()` function from the `klaR` package, which `caret` utilizes, makes an additional modification to the covariance matrix, which also has a tuning parameter γ.

$$\hat{\Sigma}_k(\lambda, \gamma) = (1 - \gamma)\hat{\Sigma}_k(\lambda) + \gamma \frac{1}{p} \text{tr}(\hat{\Sigma}_k(\lambda))I$$

Both γ and λ can be thought of as mixing parameters, as they both take values between 0 and 1. For the four extremes of γ and λ, the covariance structure reduces to special cases:

- $(\gamma = 0, \lambda = 0)$: QDA - individual covariance for each group.
- $(\gamma = 0, \lambda = 1)$: LDA - a common covariance matrix.
- $(\gamma = 1, \lambda = 0)$: Conditional independent variables - similar to Naive Bayes, but variable variances within group (main diagonal elements) are all equal.
- $(\gamma = 1, \lambda = 1)$: Classification using euclidean distance - as in previous case, but variances are the same for all groups. Objects are assigned to group with nearest mean.

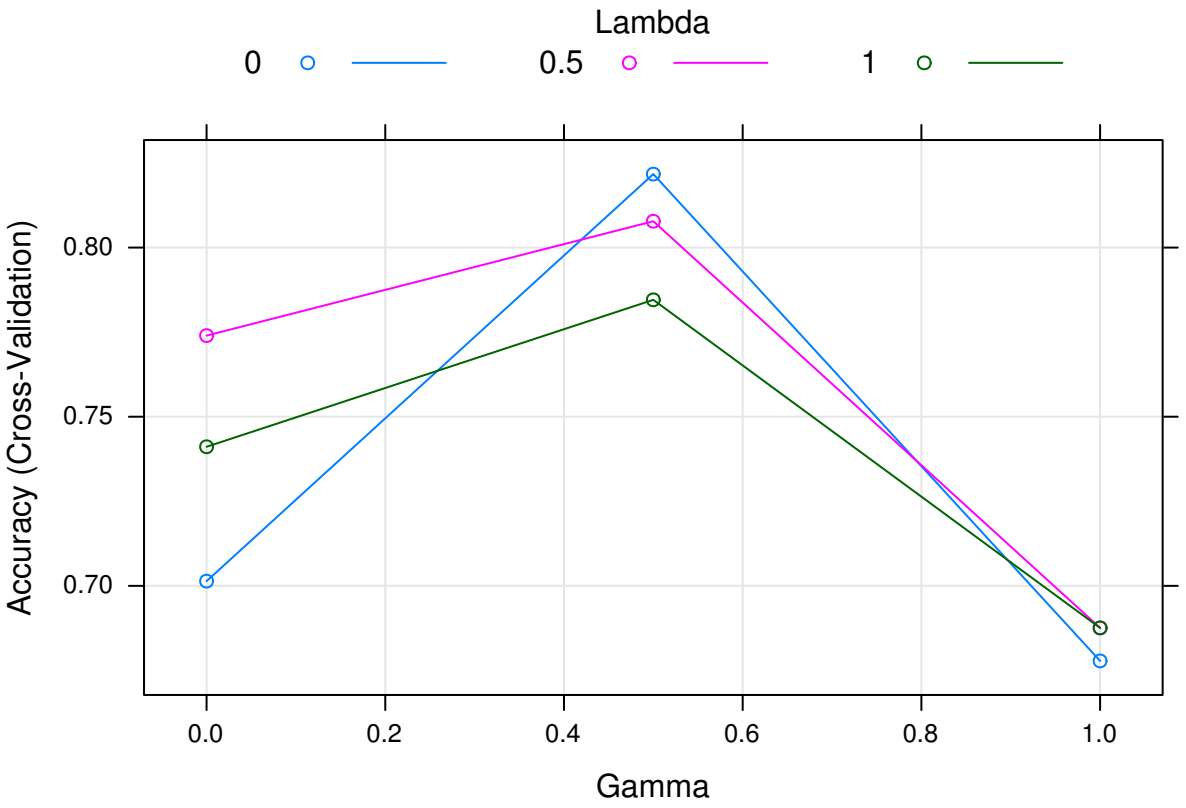
31.3 RDA with Grid Search

```
set.seed(1337)
cv_5_grid = trainControl(method = "cv", number = 5)
```

```
set.seed(1337)
fit_rda_grid = train(Class ~ ., data = Sonar, method = "rda", trControl = cv_5_grid)
fit_rda_grid
```

```
## Regularized Discriminant Analysis
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 166, 166, 167, 166, 167
## Resampling results across tuning parameters:
##
##   gamma  lambda  Accuracy   Kappa
##   0.0    0.0     0.7013937  0.3841061
##   0.0    0.5     0.7739837  0.5473212
##   0.0    1.0     0.7411150  0.4789646
##   0.5    0.0     0.8217189  0.6390489
##   0.5    0.5     0.8077816  0.6095537
##   0.5    1.0     0.7845528  0.5670147
##   1.0    0.0     0.6778165  0.3535033
##   1.0    0.5     0.6875726  0.3738076
##   1.0    1.0     0.6875726  0.3738076
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were gamma = 0.5 and lambda = 0.
```

```
plot(fit_rda_grid)
```



31.4 RDA with Random Search Search

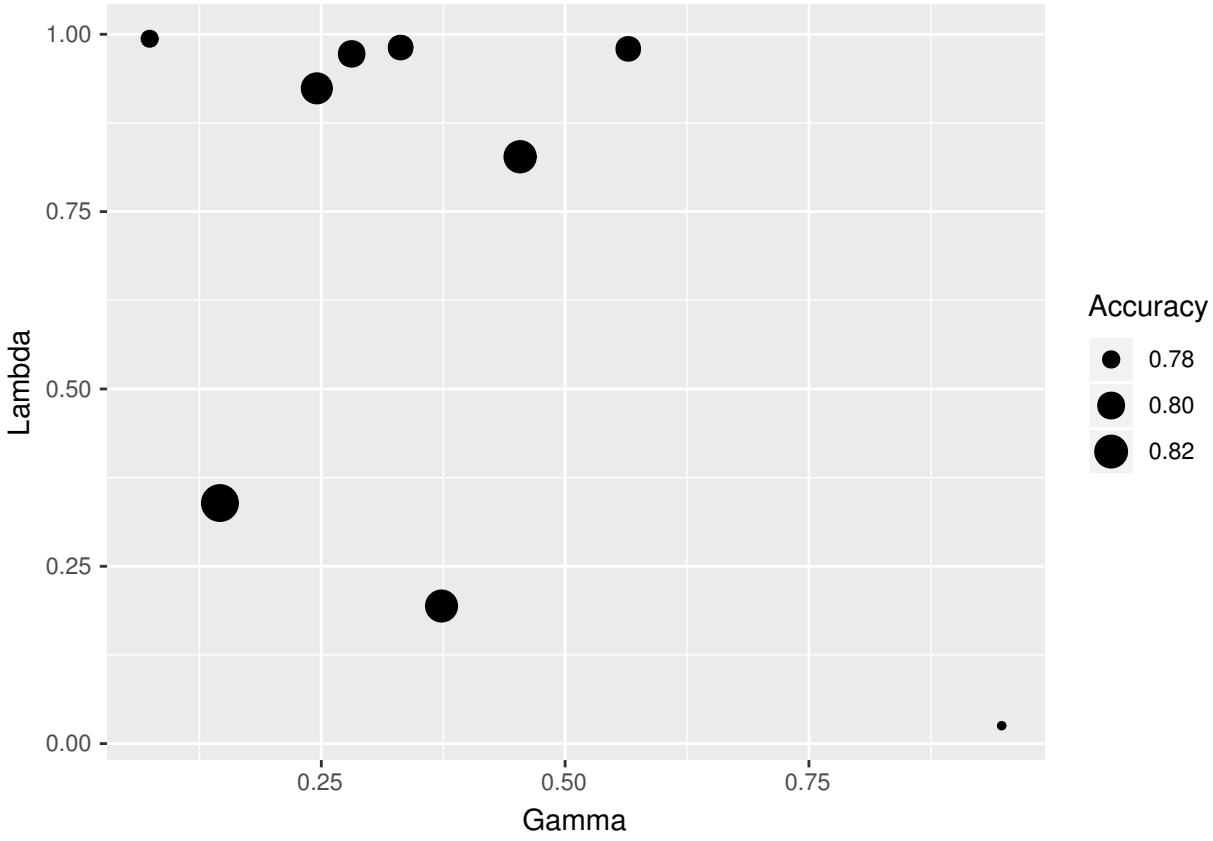
```
set.seed(1337)
cv_5_rand = trainControl(method = "cv", number = 5, search = "random")
```

```
fit_rda_rand = train(Class ~ ., data = Sonar, method = "rda",
                     trControl = cv_5_rand, tuneLength = 9)
fit_rda_rand
```

```
## Regularized Discriminant Analysis
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 166, 166, 167, 166, 167
## Resampling results across tuning parameters:
##
##   gamma       lambda      Accuracy   Kappa
##   0.07399023  0.99371759  0.7796748  0.5556869
##   0.14604362  0.33913968  0.8362369  0.6705529
##   0.24540405  0.92379666  0.8133566  0.6231035
```

```
##   0.28111731  0.97238848  0.7989547  0.5939312
##   0.33131745  0.98132543  0.7941928  0.5848112
##   0.37327926  0.19398230  0.8169570  0.6298688
##   0.45386562  0.82735873  0.8178862  0.6318771
##   0.56474213  0.97943029  0.7940767  0.5857574
##   0.94763002  0.02522857  0.7740999  0.5435202
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were gamma = 0.1460436 and lambda
##  = 0.3391397.
```

```
ggplot(fit_rda_rand)
```



31.5 Comparison to Elastic Net

```
set.seed(1337)
fit_elnet_grid = train(Class ~ ., data = Sonar, method = "glmnet",
                       trControl = cv_5_grid, tuneLength = 10)
```

```
set.seed(1337)
fit_elnet_int_grid = train(Class ~ . ^ 2, data = Sonar, method = "glmnet",
                           trControl = cv_5_grid, tuneLength = 10)
```

31.6 Results

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

```
knitr::kable(rbind(
  get_best_result(fit_rda_grid),
  get_best_result(fit_rda_rand)))
```

gamma	lambda	Accuracy	Kappa	AccuracySD	KappaSD
0.5000000	0.0000000	0.8217189	0.6390489	0.0455856	0.0926920
0.1460436	0.3391397	0.8362369	0.6705529	0.0631932	0.1255389

```
knitr::kable(rbind(
  get_best_result(fit_elnet_grid),
  get_best_result(fit_elnet_int_grid)))
```

alpha	lambda	Accuracy	Kappa	AccuracySD	KappaSD
0.4	0.0065641	0.8034843	0.6041866	0.0645470	0.1297952
0.1	0.0243225	0.8418118	0.6809599	0.0539204	0.1088486

31.7 External Links

- Random Search for Hyper-Parameter Optimization - Paper justifying random tuning parameter search.
- Random Hyperparameter Search - Details on random tuning parameter search in `caret`.

31.8 RMarkdown

The RMarkdown file for this chapter can be found **here**. The file was created using `R` version 3.5.1 and the following packages:

- Base Packages, Attached

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

- Additional Packages, Attached

```
## [1] "klaR"    "MASS"    "glmnet"  "foreach" "Matrix"  "caret"   "ggplot2"
## [8] "lattice" "mlbench"
```

- Additional Packages, Not Attached

```
##  [1] "Rcpp"         "lubridate"    "class"        "assertthat"
##  [5] "rprojroot"    "digest"       "ipred"        "mime"
##  [9] "R6"           "plyr"         "backports"    "stats4"
## [13] "e1071"        "evaluate"     "highr"        "pillar"
## [17] "rlang"        "lazyeval"     "rstudioapi"   "data.table"
## [21] "miniUI"       "rpart"        "combinat"     "rmarkdown"
## [25] "labeling"     "splines"      "gower"        "stringr"
## [29] "questionr"    "munsell"      "shiny"        "compiler"
## [33] "httpuv"       "xfun"         "pkgconfig"    "htmltools"
```

```
## [37] "nnet"         "tidyselect"   "tibble"       "prodlim"
## [41] "bookdown"     "codetools"    "later"        "crayon"
## [45] "dplyr"        "withr"        "recipes"      "ModelMetrics"
## [49] "grid"         "xtable"       "nlme"         "gtable"
## [53] "magrittr"     "scales"       "stringi"      "reshape2"
## [57] "promises"     "bindrcpp"     "timeDate"     "generics"
## [61] "lava"         "iterators"    "tools"        "glue"
## [65] "purrr"        "survival"     "yaml"         "colorspace"
## [69] "knitr"        "bindr"
```

Chapter 32

Support Vector Machines

TOOD: This chpater is currently empty to reduce build time.