Stat2

Building Models for a World of Data

R Companion

Ann R. Cannon
Cornell College

George W. Cobb
Mount Holyoke College

Bradley A. Hartlaub
Kenyon College

Julie M. Legler
St. Olaf College

Robin H. Lock
St. Lawrence University

Thomas L. Moore
Grinnell College

Allan J. Rossman
California Polytechnic State University

Jeffrey A. Witmer
Oberlin College

W. H. Freeman and Company
New York

ISBN-13: 978-1-4641-0268-4
ISBN-10: 1-4641-0268-6

Printed in the United States of America

First printing

W. H. Freeman and Company
41 Madison Avenue
New York, NY 10010
Houndmills, Basingstoke RG21 6XS, England

www.whfreeman.com

Contents

Introduction

R is an open-source computer language tailored toward data analysis and presentation. R runs on Windows, Mac, or Linux platforms. R is powerful and flexible, and it differs from many software packages in that it uses a command-line interface rather than a gui or menu-driven interface. In saying this, we acknowledge that there are presently projects ongoing that give the user an optional gui environment, but we present R here in its pure form, as the gui options can be learned independently. Underlying any implementation of R is the command-line interface.

This document will follow the order of the textbook, so that you may read it along with reading the book. That is why we call it a companion.

Note about figure, example, and section numbering: In this *Companion* we will replicate many examples and figures that appear in the text. In all such cases, we will use the same figure or example number as appears in the text. For figures or examples that are unique to the *Companion* we will use a figure or example identifier that is of the form `chapter.letter`. For example, Figure 1.2 in the *Companion* is also Figure 1.2 in the text; Figure 1.A is a figure unique to the *Companion*. Where possible, we will also make section numbers in the *Companion* match those in the text.

We will occasionally refer to two books that are useful reference guides to R. They are:

- *The R Book*, by Michael J. Crawley, 2007: Wiley, Chichester, England.
- *An Introduction to R*, by W. N. Venables and D. M. Smith and the R Development Core Team, 2010: at the R website, http://cran.r-project.org/.

When we refer to these books we will use notation such as Crawley[page number] or RIntro[page number]. More generally, you will find these books to be useful references as you learn R.

CHAPTER -1

R Basics

(*Note:* The text begins with Chapter 0, but we begin here with Chapter -1 where we introduce basic R concepts before getting to material in the text.)

-1.1 Arithmetic and Assigning Values to Variables

How you enter the R package will depend upon whether you are on a PC, Mac, or Linux machine. Once inside R, you will work with R's prompt which is the `>` symbol. To the right of this prompt goes what you type. To complete a command, you strike the "enter" key. So, for example, here is a small piece of R input and output:

```
> 1 + 4
[1] 5
>
```

The `>` was sitting on the screen, we typed in the `1 + 4` followed by the enter key, and then R returned the `[1] 5` and the next `>` prompt. When you type an incomplete line into R, R will recognize the incompleteness and return a `+` for a prompt instead of the `>`, which will be your signal to complete the line.

Generally spaces in mathematical expressions are optional and R ignores them; thus `1 + 4` is the same as `1+ 4` is the same as `1+4`.

Arithmetic operations are `+`, `-`, `*`, `/`, and `^` for (respectively) addition, subtraction, multiplication, division, and exponentiation; order of operations is as you learned in school.

While using R for basic arithmetic is fine, we mostly use it to analyze data. R stores data and other information in various types of what are called *objects*. Of the several types of objects for data storage, the most common are *vectors*, *matrices*, and *data frames*. Data frames are special to R; they are similar to matrices but with some special properties that make them the most important

of the data storage objects.

The R language is built around *functions* that take objects as inputs and create other objects as outputs. Usually we will want to store objects—be they data storage objects or function output objects—into named variables. Thus, output from one function call can become the input for another function call. As simple as it seems, it is this capability to store any object that is one of R's most powerful features.

We begin with vectors. As in mathematics, a vector is an ordered set of numbers or an ordered set of characters. Here is a simple example, followed by a blow by blow explanation.

```
> x <- c(10, 3, 3, 4)
> x + 4
[1] 14  7  7  8
> y <- max(x)
> y
[1] 10
> x^2
[1] 100   9   9  16
```

To assign a value to a variable name we use the assignment operator which is `<-`. (*Note:* The `=` sign may also be used for assignment. You will see both in this book. The `<-` makes clear that there is a direction to the assignment operation: The object on the right is being assigned to the variable named on the left. The `=` saves you a key stroke each time.)

Here is a line by line explanation of the example. First, remember that the user is typing what comes after the $>$ on the line. Lines beginning with [1] are R's responses to a user's line. Notice that assignment lines (such as the first line and the fourth) do not result in output. To see output requires a line that either just asks for the contents of a variable (e.g., the line $> y$) or makes a calculation without an assignment (e.g., $> x + 4$).

In the first line, the concatenate function `c` builds up a vector of length 4 containing the numbers 10, 3, 3, 4. The next line defines a vector `x + 4` that adds 4 to each component of the vector x; the third line beginning with `[1]` is R's response to the second line, that is, the vector containing elements 14, 7, 7, and 8. The fourth line assigns to a variable named `y` the maximum of the vector x. Since the maximum element of the vector is 10, `y` is a variable whose value is 10. The seventh line asks for the vector `x` to be squared. The squaring occurs element by element and the result is given in line 8.

-1.2 Command Line Interface

A nice feature of R over point-and-click software is that one can write R code into a text file and execute the R code. For example, if you copy and paste the following lines into R—paste alongside

the most recent > prompt—then R executes those lines. Try it! That is, type these 4 lines into a text file and copy and paste them into the R session window. The result will be that vectors `x` and `y` are created, the number 10 is output since 10 is the maximum value of the vector `x` and the vector `[1] 86 2 2 8` will be printed out as the result of that last line. (*Note:* We intentionally left off the > prompt from these lines, so you could copy and paste them easily if you have an electronic document before you. In most cases, we will include the > prompt with R code, so cutting and pasting will require your excluding it.)

```
x <- c(10, 3, 3, 4)
y <- x+4
max(x)
x^2 - y
```

Because of this executable text feature of R, one can more easily retain what operations you are asking R to do and you can easily repeat or modify existing code. Especially when using this feature of R, comments can be very useful ways to document what the code is doing, either for someone else's benefit or your future benefit. R uses the # symbol for comments. Any text on a line that appears after that symbol is ignored by R, so it is there solely to aid you or a future user of your code. Here is the previous code with a couple of comments added. The code is exactly the same with regard to what R does.

```
# the following code illustrates some basic R syntax.

x <- c(10, 3, 3, 4)
y <- x+4  #Add 4 to each element of x
max(x)
x^2 - y  # keep in mind with R arithmetic on vectors, the operations
# occur coordinate-wise.
```

-1.3 Working with Data Objects

Entering Data

One often builds small data objects through a sequence of simple commands. We provide some examples with the R code below. We annotate pieces of the code with explanations inside R remarks (headed with #). *Note:* There are hundreds of R commands, most of which you will never use. You are not expected to memorize the commands we show in the next few lines—and you may never use some of these particular commands—but we want to show you the flavor and versatility of R.

```
# First create a vector x, length 6, with mean 4.166667 and SD 1.940790.
> x <- c(1, 3, 4, 6, 5, 6)
> x
[1] 1 3 4 6 5 6
> mean(x)
[1] 4.166667
> sd(x)
[1] 1.940790

# Next create matrix y from the vector x, making it into 3 rows (and
# thus, 2 columns).
> y <- matrix(x,nrow=3)
> y
     [,1] [,2]
[1,]    1    6
[2,]    3    5
[3,]    4    6

# Use the apply function to sum row by row (first command) ...
# and then column by column (second command).
> apply(y,1,sum)
[1]  7  8 10
> apply(y,2,sum)
[1]  8 17

# Create matrix z from the vector x, again with 3 rows and 2 columns,
# but this time reading in the numbers by rows rather than the default of
# by columns.
> z <- matrix(x,nrow=3,byrow=TRUE)
> z
     [,1] [,2]
[1,]    1    3
[2,]    4    6
[3,]    5    6
```

Besides very basic building of vectors and matrices, this code illustrates the `apply` function, a function that operates on a matrix (or more generally an array of any dimension) and performs the function given in the third argument location. The second argument of `apply` is a 1 if the operations are done row by row and a 2 if column by column.

Now, let's talk about *data frames*. We are going to take the matrix `z` and pretend it represents two pieces of information for each of three children: Emily, Tim, and Jeff. The code below creates a vector called `name` and a matrix called `my.mat`. Notice the use of the `cbind` function to combine

column-wise the vector of names `name` with the numeric matrix `z`. We use the *query* functions `is.matrix` and `is.data.frame` to verify that `my.mat` is a matrix and is not a data frame. Finally, we print out `my.mat`.

```
> name <- c("Emily", "Tim", "Jeff")  # create a vector of names
> my.mat <- cbind(name, z)  # cbind means combine column-wise
> is.matrix(my.mat)  # is.matrix is a query function ...
[1] TRUE             # ... and we learn that my.mat is a matrix
> is.data.frame(my.mat)  # (another query function)
[1] FALSE  # ... but it is not a data frame.

> my.mat   # print out the matrix my.mat
     name
[1,] "Emily" "1" "3"
[2,] "Tim"   "4" "6"
[3,] "Jeff"  "5" "6"
```

Next, we print out the second column of the matrix (using `my.mat[,2]`). Notice the quotation marks around the numbers. This means that R has interpreted all values in the matrix as character values, rather than numbers. This coercion of numbers to characters happened because matrices are only allowed to be all numbers, or all characters, or all some other type of data. Since R recognized the first column as character data, all data had to be character. Thus in the second command, when we attempt to take the mean value of the second column, R gives us an error message.

```
> my.mat[,2]
[1] "1" "4" "5"

> mean(my.mat[,2])
[1] NA
Warning message:
In mean.default(my.mat[, 2]) :
  argument is not numeric or logical: returning NA
```

Data frames are similar to matrices, but they allow both character and numeric variables. So we proceed to construct a data frame, `my.data`, using the `data.frame` function. Now, we can take the mean of the second column.

```
> my.data <-data.frame(name,z)
> is.data.frame(my.data)
[1] TRUE

> mean(my.data[,2])
[1] 3.333333
```

Now, we are going to invent a story. We will pretend the second column is a variable giving the number of teeth each child had sprouted by age 6 months. We name this variable `Teeth`. We will pretend the third column gives the month in which the child first crawled. We redefine the first variable to be `Name`. The function call `names(my.data)` assigns the vector of names as the names of this little (fictitious) data set.

```
> names(my.data) <- c("Name","Teeth","Crawl")
> my.data
   Name Teeth Crawl
1 Emily     1     3
2   Tim     4     6
3  Jeff     5     6
> names(my.data)
[1] "Name"  "Teeth" "Crawl"
```

The next section of code is important for much we will do later. Suppose we want the mean number of teeth. Notice that the command `mean(Teeth)` produces an error message. This is because we have to refer to that variable by the longer name `my.data$Teeth`. This necessity of the long-winded variable name can be cumbersome, more so if our keyboard skills are meager. So R provides a short-cut through the `attach` function. The `attach(my.data)` function call tells R to bring up the data frame `my.data` and all variable references can now be made without the cumbersome prefix.

```
> mean(Teeth)
Error in mean(Teeth) : object 'Teeth' not found
> mean(my.data$Teeth)
[1] 3.333333

> attach(my.data)
> mean(Teeth)
[1] 3.333333
```

Some Technicalities About `attach` and the *search path*

Note: We include this section on technicalities because at some point you may find it useful. But in your initial learning of R you can get by with skimming this section.

The effect of the function call `attach(my.data)` is to place the data frame `my.data` onto the system's *search path*. The search path is a sequence of objects or folders where R will search for a referenced object such as a variable. To see how this works we consider the code below. Here is the explanation. The code `search()` asks R for the current search path, that is for the sequence of directories or folders or data frames that R will search through whenever you require it to use or find an object. The first in the list is `.GlobalEnv` which refers to the current workspace; other folders follow and are built into R. The next line of code entered is the call `Teeth`. Notice from the

error message that R cannot find a variable called `Teeth` and that is because Teeth only exists as a variable in the data frame `my.data` and this data frame is not on the search path. The third line of code we execute rectifies the situation. By typing the function call `attach(my.data)` we see by the fourth line `search()` that the `my.data` data frame has now been placed in position 2 of the search path. Next, we subsequently enter `Teeth` again; this causes R to look for `Teeth` along the search path. There is no such variable in the first place looked (that is, `.GlobalEnv`), so R looks then in the second place, which is `my.data`, where it finds `Teeth` and prints it out.

To continue the lesson, we now create a second variable called `Teeth`, a vector of the consecutive integers from 1 to 10. Now, typing `Teeth` induces a search starting with the main workspace, `.GlobalEnv`, which results in R typing out the new vector `Teeth`. The version of `Teeth` in the main workspace *masks* the version inside `my.data`. We could print out the variable `my.data$Teeth` (i.e., the three numbers 1, 4, and 5) using the `get` function and putting 2 in the `pos` argument; that is looking in position 2 of the search path, the position occupied by `my.data`. (We will rarely use the `get` function, but this section was, recall, advertised as technicalities.)

When we are finished working with our data frame `my.data` we type `detach()`, which removes it from the search path.

```
> search()  # Ask R for the current search path.

[1] ".GlobalEnv" "package:stats" "package:graphics"
          "package:grDevices" "package:utils"
[6] "package:datasets"  "package:methods"
          "Autoloads"   "package:base"

> Teeth  # Since my.data is not on the search path
         # R cannot find it; hence this error message:
Error: object 'Teeth' not found

> attach(my.data)  # This command puts my.data onto
                   # the search path, which ...
> search()         # ... this command verifies.

 [1] ".GlobalEnv" "my.data" "package:stats"
            "package:graphics" "package:grDevices"
 [6] "package:utils" "package:datasets" "package:methods"
            "Autoloads" "package:base"
> Teeth    # Now R can find the vector Teeth.
[1] 1 4 5

> Teeth <- 1:10  # We create a second vector Teeth
                 # which resides in the main workspace
```

```
                # called .GlobalEnv, which is in
                # position 1 of the search path.

> Teeth         # Thus, when we ask for Teeth, R finds
                # the one in .GlobalEnv first.

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

> get("Teeth",pos=2)  # If we want the other Teeth,
                      # we must specify position in the
                      # search path.
[1] 1 4 5

> detach()
> search()

[1] ".GlobalEnv" "package:stats" "package:graphics"
             "package:grDevices" "package:utils"
[6] "package:datasets"  "package:methods"
             "Autoloads"  "package:base"
```

Getting an External Dataset and Extracting Pieces of It

Usually, we will be working with larger data frames that are stored externally. There are a variety of R functions for reading in such data; type `help(read.table)` to get an overview. We will illustrate one we will favor for this document, the version called `read.csv`, which reads external comma-separated-variable datasets.

Here is an example. We obtain the dataset described in Exercise 0.11 of the text. The variables, named below, are answers to questions on a survey given to students. The `dim(Day1Survey.df)` tells us that there are 43 cases and 13 variables in the dataset. We use the `names` function to print out the names of the 13 variables. *Note:* Here we have assigned the name of the data frame as `Day1Survey.df`. There is no technical meaning to the suffix `.df` at the end of this name; we simply use it to remind outselves that this object is a data frame.

```
Day1Survey.df <- read.csv(file=file.choose())
> dim(Day1Survey.df)
[1] 43 13
> names(Day1Survey.df)
[1] "Section"  "Class" "Sex" "Distance"  "Height"  "Handedness"  "Coins"
[8] "WhiteString" "BlackString" "Reading"  "TV"  "Pulse"   "Texting"
```

We now will use this dataset to illustrate important R operations for extracting pieces of a data set. Skill in manipulating datasets is invaluable, takes time to learn, but is well worth learning.

First, attach the dataset, so we can easily refer to variable names.

```
attach(Day1Survey.df)
```

The first command line below illustrates extracting sections of data using the [ ] notation. The first entry in the pair refers to rows, the second to columns. We have asked R to print out the first row and columns 3, 4, and 6, and we find that the first student in the list was a female, whose home is 400 miles from campus, and she is right-handed. The second command shows that we can use the names rather than the column numbers, but names are in quotes. If we leave blank the rows selection then we get all rows; if we leave blank the columns selection then we get all columns. The blank argument acts as a "wild card." For example, the third command asks for all rows, since the first entry (before the comma) is blank; we are thus given the sex of each of the 43 cases.

```
> Day1Survey.df[1,c(3,4,6)]
  Sex Distance Handedness
1   F      400      Right

> Day1Survey.df[1,c("Sex","Distance","Handedness")]
  Sex Distance Handedness
1   F      400      Right

> Day1Survey.df[,3]
 [1] F F F M F M F M M M M M M F M F M M M M F M M F M M F M
 [29] F M F F M M F M M M M F F M F
Levels: F M
```

The following command line is a logical operation asking whether each case has value "M", that is, it asks whether a case is male or not. The result is a sequence of 43 TRUE or FALSE values. Notice the first three cases are female, followed by a male, then a female, then a male, etc. The next command line `males.df <- Day1Survey.df[Sex=="M",]` creates a subset of the data, a new data frame called `males.df` that contains 26 cases and all 13 variables. Finally, we construct a dataset called `TallTV.males.df` consisting of those males who are 72 inches tall or taller but only selecting out their Height and TV values. Notice the use of the ampersand (`&`) to indicate the logical operator AND in order to subset out those cases that are both male AND at least 72 inches tall.

```
> Sex=="M"
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
 [11] TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
 [19] TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE
```

```
 [30] TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
 [37]  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

```
> males.df <- Day1Survey.df[Sex=="M",]
> dim(males.df)
[1] 26 13
```

```
> TallTV.males.df <- Day1Survey.df[Sex=="M" & Height>=72,c("Height","TV")]
> dim(TallTV.males.df)
[1] 12  2
```

-1.4 R Functions: Built-In and User-Defined

R is all about functions. Functions accept numbers, vectors, matrices, and other objects as inputs and give outputs that will be an R object of some kind. Functions are of two types: functions already defined in the R language and functions you, the user, define.

Built-In Functions

R provides you with a large collection of built-in functions for doing numerical and vector mathematics and logical operations. We will introduce some functions here and others as we work our way through the text.

R provides some familiar mathematical functions such as `log`, `exp`, and `sqrt`. These three are used often in statistics. (There are many other functions, including trigonometric functions.) Here are some examples:

```
> exp(1)
[1] 2.718282
```

```
> log(exp(4))
[1] 4
```

```
> sqrt(20)
[1] 4.472136
```

Notice that `log` refers to natural (base e) log. There are also the functions `log10` and `log2` for base-10 and base-2 logs. But arbitrary bases are possible using the `log` function. To see this, we use one of the more helpful of R functions, the `help` function, which the following illustrates:

```
> help(log)

[Much of the R help message is expunged here]

log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)
```

The `help` message causes R to produce a help message explaining the function in question. It pays learning to read help messages, even though they often give more information than one needs at the moment. In this case, we expunged much of the superfluous (for this discussion) output. The main point is that the top line of syntax is not as simple as `log(x)`. The "x" in the command is an argument, but R tells us there is a second argument, called "base." We now digresss to arguments of functions.

Arguments to Functions

In the command `log(20)`, the "20" is the argument to the function. But, as we know, functions may have multiple arguments and—as the help message informed us—the `log` function does. The help message tells us that the first argument (the "x") is the value we will take the logarithm of. The second argument is the number we will use as the base for the logarithm. Here is a sequence of lines that will illustrate the function and some conventions about how arguments work.

```
> log(8)  # the base-e log of 8
[1] 2.079442
> log(8,base=exp(1)) # same result as the default base is e
[1] 2.079442
> exp(1)
[1] 2.718282 # just to see that exp(1) really is e
> log(8,base=2)  # now log of 8, but base 2
[1] 3
> log(8,2)  # one can define arguments by position instead
[1] 3
> log(2,8)  # but the log of 2 to base 8 is different
[1] 0.3333333
```

The first command line (meaning a line beginning with a >) asks for the natural log (base e log) of 8. (You could check this with a hand-held calculator.) The second command line does the same thing, since the base given in the second argument—`exp(1)`—is just the base of the natural log, $e = 2.718282....$ The third command line asks for the base-2 log of 8, which is 3. The fourth command line illustrates the principle that arguments can be *passed by name* or they can be *passed by position*. As the `help` message told us, the default position of the `base` argument is position 2, so if we put the base in that position, the string "base=" is superfluous.

Finally, that fifth line of R code asks for the base-8 log of 2, which is 1/3.

User-Defined Functions

Users can define functions. Any time you find yourself doing a similar sequence of commands over and over you should consider writing a function for the task. Functions can be very simple or quite complicated. We will introduce some user-defined functions throughout this companion, but we will illustrate here with a simple example. If you copy the first four lines below into R, you will have defined a function called `mymean`, which allows the user to compute a trimmed mean of a vector of nonmissing values. The main argument, the first one, is the vector, generically called `x` in the function definition. There is a second argument, `trim`, that tells how many items of the vector to trim (the same number trimmed high and low). We give a default value for `trim`, which means that if the user doesn't give a value to the `trim` argument the function will trim off no values, thus computing the usual mean value of the vector.

We follow the function definition with examples of its use. We create a vector `z` of length 7. The first call to `mymean` computes the straight mean value, 14. Then we call the function, but trim one value off each end; answer is 4. Then we trim 2 off each end; now the mean is taken of the middle 3 values, but this is still 4.

```
mymean <- function(x,trim=0) {
              y <- sort(x)
              mean(y[(trim+1):(length(y)-trim)])
}

> z <- c(1:6,77) # to the vector 1, 2, ..., 6, we append 77

> mymean(z)  # default is no trimming, so the usual mean
[1] 14

> mymean(z,1) # trim off the max and min; now the mean is 4
[1] 4

> mymean(z,2) # trim off 2 high and low; mean of middle 3 is still 4.
[1] 4
```

We will illustrate other user-defined functions throughout the companion.

CHAPTER 0

What is a Statistical Model?

Example 0.6: Financial Incentives for Weight Loss: Data analysis with two samples.
In the commands that follow, we obtain the dataset **WeightLossIncentive4**,[1] `attach` it, and produce a comparative dotplot. (Recall that `attach` puts our data frame in the search path so that we can refer to variable names easily, without typing the cumbersome prefix `WeightLoss.df`.) Here is how the code works. After obtaining the data frame `WeightLoss.df`, we use the `str` function to see its structure. The first variable, `WeightLoss`, is numerical. The second variable, `Group`, is a factor. The dataset contains two variables.

Notice also that the dotplot function requires calling in the R `lattice` *library*. A library is a folder or collection of R objects that someone has collected together for some purpose. The lattice library contains some special graphing functions. Notice also that the dotplots readily available with R look different from many of those in the textbook, which are from Minitab.

```
#Note:  Our convention will be to use the suffix .df for data frames.

> WeightLoss.df <- read.csv(file=file.choose())
         #obtain the data set; R allows you to search for it.

> str(WeightLoss.df)
'data.frame':   36 obs. of  2 variables:
 $ WeightLoss: num  12.5 12 1 -5 3 -5 7.5 -2.5 20 -1 ...
 $ Group     : Factor w/ 2 levels "Control","Incentive": 1 1 1 1 1 1 1 1 1 ...

> attach(WeightLoss.df)  # to allow us to more easily refer to variable names

library(lattice)  # calls up the lattice library, which contains
                  # some graphing functions we need.

xyplot(WeightLoss ~ Group) # produces comparative dotplots
```

[1] There is another dataset called **WeightLossIncentive**, but it contains some unwanted missing values.

The incentivized treatment group does tend to lose more weight than the control group. We can add summary statistics to this picture with the following code. The `tapply` function is similar to the `apply` function we encountered in the *Basics* chapter. In `tapply` the second argument acts as a "by" variable for the first variable, so that the function listed in position 3 (`length` in the first instance; `mean` in the second) is applied to the first variable using the subsets of the data created by values of the second variable.

Hence `tapply(WeightLoss, Group, mean)` finds the mean of the `WeightLoss` variable separately for each value of `Group`; there are 2 Group values (Control and Incentive) and so we get a mean for each. At this point then, `n`, `mean`, and `SD` are each vectors of length 3; the `cbind` function combines these three vectors *column-wise* into a matrix of 3 columns, each of length 2.

```
# Assuming here WeightLoss.df is attached

n <- tapply(WeightLoss,Group,length)
mean <- tapply(WeightLoss,Group,mean)
SD <- tapply(WeightLoss,Group,sd)
cbind(n,mean,SD)  #combine the 3 vectors column-wise.
```

The resulting output:

```
           n      mean       SD
Control   19  3.921053 9.107785
Incentive 17 15.676471 9.413988
```

We tidy up the output for more graceful communication to the reader, using the `round` function. The code and output follow.

```
round(cbind(n,mean,SD),2)  # We choose rounding to 2 decimal places.
```

```
           n  mean   SD
Control   19  3.92 9.11
Incentive 17 15.68 9.41
```

With graphical and summary comparisons that suggest the Incentive group outloses the Control group, we could follow with a two-sample t-test to ascertain whether the observed difference is statistically significant. To do this we use the command `t.test(WeightLoss ~ Group)`.[2] We include the code below, which shows the difference, in favor of the Incentive group, to be highly statistically significant.

[2]There are two versions of the two-sample t-test: the classical "pooled" version that assumes equal population variances or the more robust, but approximate t-test, often called Welch's test. The `var.equal` argument lets us make this choice, the default `var.equal=F` being the Welch test. Using the argument `var.equal=T` would get the pooled t-test.

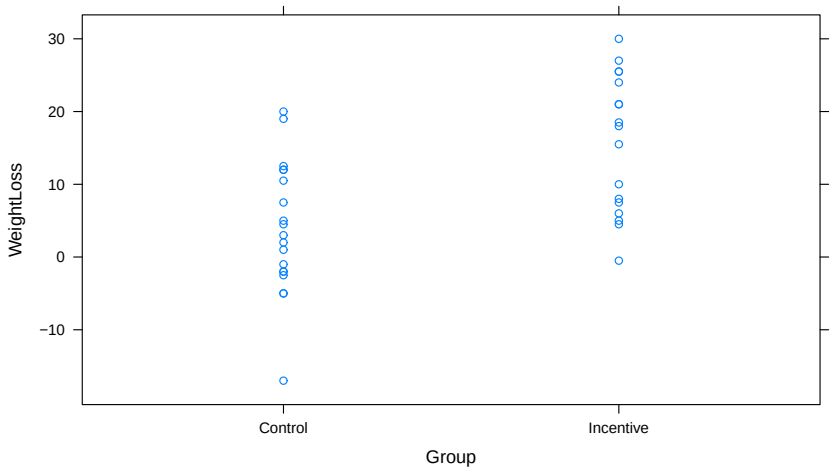


Figure 0.2: Dotplot of WeightLoss versus Group (page 8)

```
t.test(WeightLoss ~ Group)

        Welch Two Sample t-test

data:  WeightLoss by Group
t = -3.7982, df = 33.276, p-value = 0.0005889
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -18.05026  -5.46058
sample estimates:
  mean in group Control mean in group Incentive
               3.921053                15.676471
```

Beyond the Data Analysis: Fit and Residual

R has built-in ways to find fitted values and residuals for the two-sample model discussed in the text. Instead, we will take a somewhat more labor-intensive sojourn into R to compute these values, our purpose being two-fold: (1) to gain some more practice in R basics, and (2) to get a more visceral feel for what the fitted values and residuals actually mean.

Here is an explanation of the code; to really understand what is going on, it is instructive to reproduce the code on your own, and to produce some of the pieces of the code suggested in what follows. We first define a binary variable called `Group.dummy`. Recall that `Group` is a binary variable with *character* values `Incentive` and `Control`. The logical expression `Group=='Incentive'` produces

a vector of the same length as `Group` (i.e., 36) that has value TRUE or FALSE, depending on whether that entry of `Group` has value `Incentive` or `Control`. If you are working through this example in your own R session, a useful check on your understanding would be to type in just the piece of code `Group=='Incentive'`. Because we want the `Group.dummy` variable to be numeric and not character, we use the `as.numeric` function, which converts TRUEs to 1's and FALSEs to 0's. Next, we create two mean values `mu.1` and `mu.2`, the means of the Control and Incentive groups, respectively. Then, we define vector `Fit` (length 36 again), which has value `mu.1` for Control cases and `mu.2` for Incentive cases.

```
Group.dummy <- as.numeric(Group=="Incentive") # Create dummy variable for Group
                                              # 1=Incentive, 0=Control

mu.1 <- mean(WeightLoss[Group.dummy==0])
mu.2 <- mean(WeightLoss[Group.dummy==1])

Fit <- mu.1*(1-Group.dummy) + mu.2*Group.dummy
Resid <- WeightLoss-Fit

cbind(WeightLoss, Fit, Resid)

      WeightLoss       Fit       Resid
 [1,]       12.5  3.921053   8.5789474
 [2,]       12.0  3.921053   8.0789474
 [3,]        1.0  3.921053  -2.9210526
 [4,]       -5.0  3.921053  -8.9210526
 [5,]        3.0  3.921053  -0.9210526

(output suppressed after the 5th row)
```

Understanding these R commands should enhance your understanding of R syntax. In the construction of the two sample means, `mu.1` and `mu.2`, we make use of a dummy variable for the Group factor.

It is also instructive to construct "by hand" the t-statistic for the two-sample t-test above. The code for this calculation follows.

The `sd` function computes a sample standard deviation. The `qqnorm` function computes a normal probability plot and the `qqline` function adds a line to this plot in order to help assess the straightness of the normal plot.

```
> s1 <- sd(WeightLoss[Group=="Control"])
> s1
[1] 9.107785
> s2 <- sd(WeightLoss[Group=="Incentive"])
> s2
[1] 9.413988

> n1 <- length(WeightLoss[Group=="Control"])
> n1
[1] 19
> n2 <- length(WeightLoss[Group=="Incentive"])
> n2
[1] 17

> t.stat <- (mu.1 - mu.2)/sqrt(s1^2/n1 + s2^2/n1)
> t.stat
[1] -3.911902
 # NOTE:  This result agrees with the previous output.
```

We re-produce Figures 0.3 (page 10) and 0.4 (page 10) using the code below.

```
> xyplot(Resid ~ Group)

> ControlResidual <- Resid[Group=="Control"] # Control residuals
> IncentiveResidual <- Resid[Group=="Incentive"] # Incentive residuals
> qqnorm(ControlResidual, ylab="Control Residual", main= "") #normal plot
> qqline(ControlResidual) # adds line

> qqnorm(IncentiveResidual, ylab="Incentive Residual", main= "") # normal plot
> qqline(IncentiveResidual) # add the line
```

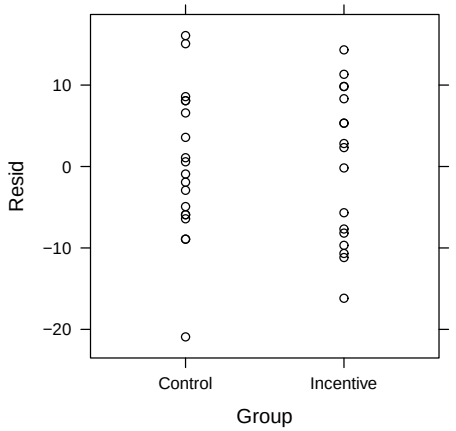
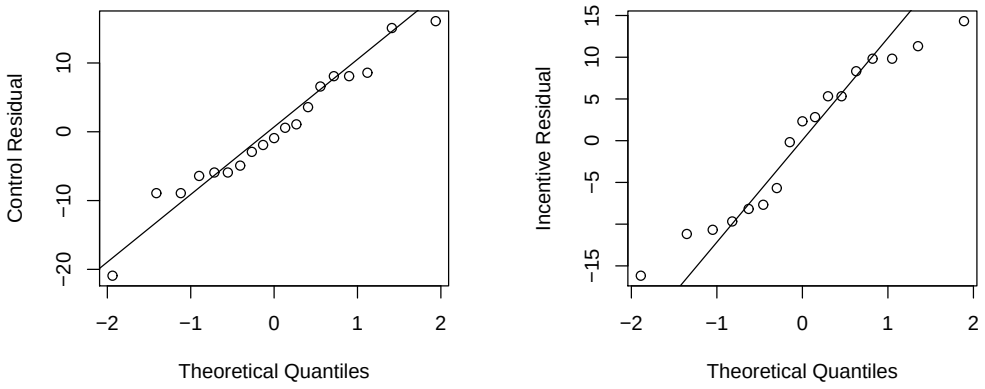


Figure 0.3 (page 10): Residuals from Group Weightloss Means



(a) Normal plot of Control residuals

(b) Normal plot of Incentive residuals

Figure 0.4 (page 10) Normal Probability Plots for Residuals of Weightloss

CHAPTER 1

Simple Linear Regression with R

1.1 The Simple Linear Regression Model

Example 1.1 (page 25): Porsche Prices

We now proceed to use R with the regression chapters of the text . We begin with the example on Porsche Prices. Here is the code to obtain and plot the data.

```
Porsche.df <- read.csv(file=file.choose())  # Get the data.
dim(Porsche.df)    # Do some basic checks that we have the right data.
[1] 30  3          # 30 rows and 3 columns looks right.
names(Porsche.df)  # The names that follow agree with the book.

[1] "Price"   "Age"     "Mileage"

attach(Porsche.df)  # Remember, we use attach to refer more easily to variable names.
plot(Mileage, Price)  # The basic scatterplot of interest.
```

To fit a simple linear regression model we use R's `lm` function. The `lm` function is a function for fitting the general linear model to a dataset. The presumption is that there is a quantitative response variable and one or more predictors, which can be quantitative or categorical.

In this call to the `lm` function, we store the model output into a new variable named `Porsche.lm1`. We will soon realize that R has stored more information into `Porsche.lm1` than we might initially suspect.

```
Porsche.lm1 <- lm(Price ~ Mileage)
abline(Porsche.lm1)
```

We now add a regression line to the scatterplot using R's `abline` function. The `abline` function is a general function for adding lines with specified slope and intercept to an exisiting plot. The standard form of the command uses two arguments—the intercept in position 1 and the slope in

position 2. However in our call to `abline` we do not directly give intercept and slope but instead hand `abline` the name of our fitted simple linear regression model, and R magically extracts from this object the required intercept and slope. This flexibility of dealing with function arguments—that is, doing the right thing based upon recognizing the nature of the input—is one of R's many strengths. The scatterplot with added regression line appears in Figure 1.2.

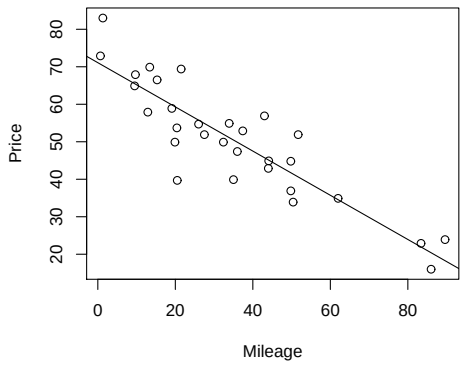


Figure 1.2 (page 29): Scatterplot of Price versus Mileage with regression line

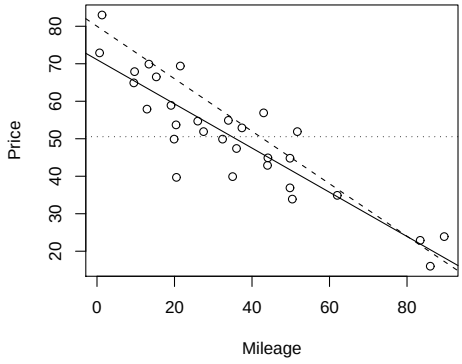
To carry this discussion a bit further, try the following R commands.

```
> abline(80, -.7, lty=2)  # Add a second, dashed line, which fits less well.
> abline(h=mean(Price),lty=3) # Add a horizontal, dotted line, at the mean Price level.
```

The first command adds a line with intercept 80 and slope -0.7, a line that runs through the data but fits the data less well than the regression line. The second command uses the `h` argument for the `abline` function, which gives a short-cut for adding a horizontal line at the level of the mean of Price. Notice that these two calls to `abline` also use the `lty` argument which stands for *line type*. There is a code for line types, where 1 is the default solid line, 2 is dashed, 3 is dotted, 4 is dot-dash, 5 is long dash, and 6 is two dash.

Unpacking an R Object: Porsche.lm1

Common R objects that we will work with a lot will be vectors and data frames. The *list* is another object type and should be thought of as an ordered collection of other objects. Think of a list as sort of a queue at a movie theater, except that members of the queue don't all have to be, say, people. The first might be a person, the second a hyena, the third a suitcase full of thousand dollar bills, the fourth an invitation to a surprise birthday party, etc. Each member of that queue then has it's own kind of structure because it is an object of a certain type. The output of a call to



Scatterplot with added lines to illustrate abline

the `lm` function, as we did above in the `Porsche.lm1 <- lm(Price ~ Mileage)` code, is a list. Consider the following sequence of R code with resulting output.

```
> is.list(Porsche.lm1)
[1] TRUE
> length(Porsche.lm1)
[1] 12
> is.vector(Porsche.lm1)
[1] FALSE
> names(Porsche.lm1)
[1]"coefficients" "residuals" "effects" "rank" "fitted.values" "assign"
[7]"qr"  "df.residual"  "xlevels"  "call"  "terms"  "model"

> Porsche.lm1[[1]]
(Intercept)     Mileage
 71.0904527  -0.5894009
> Porsche.lm1$coefficients
(Intercept)     Mileage
 71.0904527  -0.5894009
> Porsche.lm1$coef
(Intercept)     Mileage
 71.0904527  -0.5894009
> is.vector(Porsche.lm1$coef)
[1] TRUE
> length(Porsche.lm1$coef)
[1] 2
```

The first line is a query function asking R if the object `Porsche.lm1` is a list, which it is. Then we ask for its length and find it is a list of length 12. Just for fun, we asked if `Porsche.lm1` is a vector, which it is not.

It turns out that the 12 objects comprising this list collectively contain all the useful information we will need about the model, things like the coefficients (slope and intercept), the residuals, the fitted values, among others.

We then asked for the names of the 12 components of the list, which are `coefficients`, `residuals`, etc. To access a component of a list, one uses double brackets [[ ]]. The command `Porsche.lm1[[1]]` causes R to print out the vector of coefficients—intercept and slope, in that order—which comprises the first element of the list. Since it would be cumbersome to have to remember what each of the 12 components of the list stand for, we see in the `Porsche.lm1$coefficients` command that we can just as easily refer to this part with a more transparent identifier and the next line `Porsche.lm1$coef` illustrates that we don't have to write out the entire label "coefficients," but just enough of it to uniquely distinguish it from the other components of the list. The last two commands merely confirm for us that `Porsche.lm1$coef` is a vector of length 2.

Below we give some code to produce a nice table of the data along with fitted values and residuals. First is a line of R code for printing a table of the data points along with fitted values and residuals, using the `cbind` function. Notice that the first row corroborates the values given in the text. You might punch some numbers into a hand-held calculator at this point to confirm where the fits and residuals are coming from. *Note:* The notation `1:5` in that call to `cbind` asks R to print out only rows 1 through 5, since `1:5` is notation for the vector `(1,2,3,4,5)`. Note also that this command ends with "[1:5,]" which means "produce all columns (for rows 1:5)." Had we written "[1:5,2]," for example, then we would have gotten only column 2 (the Prices).

```
> cbind(Mileage,Price,Porsche.lm1$fit,Porsche.lm1$resid)[1:5,]
                                                #print first 5 rows

  Mileage Price
1    21.5  69.4 58.41833 10.981667
2    43.0  56.9 45.74621 11.153788
3    19.9  49.9 59.36137 -9.461374
4    36.0  47.4 49.87202 -2.472019
5    44.0  42.9 45.15681 -2.256811

> Porsche.lm1.table <- data.frame(cbind(Mileage,Price,Porsche.lm1$fit,Porsche.lm1$resid))
> names(Porsche.lm1.table)
[1] "Mileage" "Price"   "V3"      "V4"

> names(Porsche.lm1.table)[c(3,4)] <- c("fits", "residuals")
> head(Porsche.lm1.table)
```

```
  Mileage Price     fits residuals
1    21.5  69.4 58.41833 10.981667
2    43.0  56.9 45.74621 11.153788
3    19.9  49.9 59.36137 -9.461374
4    36.0  47.4 49.87202 -2.472019
5    44.0  42.9 45.15681 -2.256811
6    49.8  36.9 41.73829 -4.838286
```

The `cbind` function creates a matrix by concatenating the 4 columns mentioned in the argument. To tidy up this summary table of fits and residuals, we transform the matrix into a data frame using the `data.frame` function. The next line of code asks for the names of the 4 variables in the data frame, `Porsche.lm1.table`; notice that the first two are `Mileage` and `Price`, but the next two are default values given by R, namely `V3` and `V4`. We follow up this line with a line that assigns to these two variables the more transparent names `fits` and `residuals`. The `head` function is a function, similar to `str`, that gives a brief overview of the structure of a data frame. While `str` lists variables and their types and the first few values, `head` causes the first few rows of the data frame to be printed. Either `head` or `str` are useful ways to check that a data frame you have obtained or defined looks like what you think it should look like.

Finally, we note the `summary` function's effect on the `Porsche.lm1` object. The `summary` function is an example of a *generic* function, which means that the type of output it produces will depend upon the type of object used as input. In this case, the input object is a regression object. R recognizes this and produces a standard regression summary table, which looks similar to the Minitab table in the text.

```
> summary(Porsche.lm1)

Call:
lm(formula = Price ~ Mileage)

Residuals:
     Min       1Q   Median       3Q      Max
-19.3077  -4.0470  -0.3945   3.8374  12.6758

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 71.09045    2.36986   30.00  < 2e-16 ***
Mileage     -0.58940    0.05665  -10.40 3.98e-11 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 7.17 on 28 degrees of freedom
Multiple R-squared: 0.7945,     Adjusted R-squared: 0.7872
```

```
F-statistic: 108.3 on 1 and 28 DF,  p-value: 3.982e-11
```

The following example shows that if the input object were a vector instead of a regression object, R would give a different kind of summary. The summary produced is the five-number summary plus the mean value. Other R objects will result in other appropriate summaries.

```
> summary(c(1,2,3,3,4,5,10))  # Here the input is a vector of length 7.

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     2.5     3.0     4.0     4.5    10.0
```

1.2 Conditions for a Simple Linear Model: Regression Standard Error

In Example 1.4 on Porsche Prices we show that the regression standard error is 7.17. We can find this value in the output from the R command `summary(Porsche.lm1)` as "Residual standard error: 7.17 on 28 degrees of freedom."

1.3 Assessing Conditions: Diagnostic plots

The text introduces us to several diagnostic plots for regression, meaning plots involving the residuals that help us assess the reasonableness of the regression model conditions. Each is easy to implement with R using the object created by the `lm` function, in our example the object `Porsche.lm1`.

Normal Plots and Histograms of Residuals

We re-create Figures 1.8 and 1.9 using:

```
> plot(Porsche.lm1$fitted.values,Porsche.lm1$residuals)
> hist(Porsche.lm1$resid)
```

Two functions produce the normal quantile plot with superimposed line: `qqnorm` and `qqline`. Here is a salient example.

```
> qqnorm(Porsche.lm1$resid) # Plot residuals vs. theoretical normal quantiles
> qqline(Porsche.lm1$resid) # Fits a straight line to "aid the eye."
```

Since we will usually follow the normal plot with the aidful fitted line, we can create our own user-defined function to accomplish both, which we call `myqqnorm`. Type these lines into R to define the function, then try it out with the function call `myqqnorm(Porsche.lm1$resid)`. NOTE: To try out the new function, you should delete the previous graph, because `myqqnorm` is simply going to reproduce it, and you want to be clear that it has done so.

```
myqqnorm <- function(x){
            qqnorm(x)
            qqline(x)
}

> myqqnorm(Porsche.lm1$resid)  #Now, try it out, after deleting previous graph.
```

1.4 Transformations and Other Residual Plots

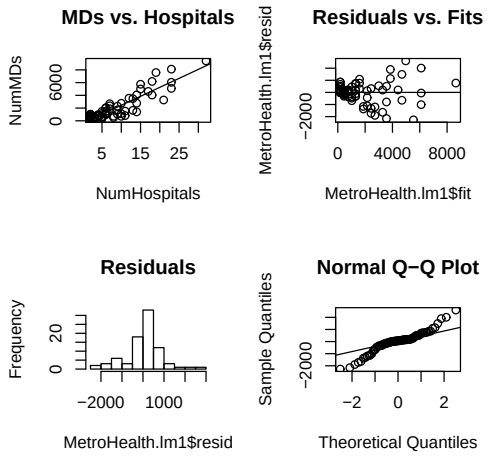
Example 1.6: Doctors and Hospitals Example

Let's use the Doctors and Hospitals example to illustrate using R to get the panoply of residual plots we use for regression diagnostics.

After obtaining the data, we see our first instance of one of R's most powerful functions. The `par` function helps us control a wide variety of graphical parameters, such as line style, plotting character, axes, color, and on and on. See Crawley[848-850] for a table of all graphical parameters. The command `par(mfrow = c(2,2))` partitions our graphics window into a 2-by-2 panel of panels, so we can get 4 graphs on a single window. Note: As a matter of style, after creating the panel of 4 graphs we use `par(mfrow=c(1,1))` to return to the default one graph per panel, so that subsequent work does not stay in the 2-by-2 mode. Note as well, that `abline(h=0)` adds a horizontal line at level $y = 0$, i.e., the x-axis.

```
> MetroHealth83.df <- read.csv(file=file.choose())  # Get the data.
> MetroHealth.df <- MetroHealth83.df[,c(1,2,4)] # extract the 3 variables
                                                # used in Chapter 1 of text
> attach(MetroHealth.df)
> par(mfrow=c(2,2))
> plot(NumHospitals, NumMDs,main="Scatterplot of MDs vs. Hospitals")
> MetroHealth.lm1 <- lm(NumMDs ~ NumHospitals) # fit model
> abline(MetroHealth.lm1) # Add line to scatterplot
> plot(MetroHealth.lm1$fit, MetroHealth.lm1$resid, main="Residuals vs. Fits")
> abline(h=0) # see comment above
> hist(MetroHealth.lm1$resid, main="Histogram of Residuals")
> myqqnorm(MetroHealth.lm1$resid)
> par(mfrow=c(1,1))  # return to single graph window
```

We have replicated the diagnostic plots from the textbook, the ones that suggest a transformation of the data. The R code below fits the square-root model and produces the diagnostic plots. (*Note:* We suppress these diagnostic plots, which you can find in the text.)



Figures 1.12(a) (page 41), 1.12(b) (page 41), 1.13(a) (page 41), and 1.13(b) (page 41): Diagnostic Plots for Metro Health Data (original scale)

```
> SqrtMDs <- sqrt(NumMDs)  # Create the square-rooted variable
> MetroHealth.df <- data.frame(MetroHealth.df,SqrtMDs) # Add it to the data frame
> dim(MetroHealth.df)  # Check that it got added
[1] 83  4
> names(MetroHealth.df)
[1] "City"         "NumMDs"       "NumHospitals" "SqrtMDs"   # This looks right

> MetroHealth.lm2 <- lm(SqrtMDs ~ NumHospitals)
> par(mfrow=c(2,2))
> plot(NumHospitals, SqrtMDs)
> abline(MetroHealth.lm2)
> plot(MetroHealth.lm2$fit, MetroHealth.lm2$resid)
> abline(h=0)
> hist(MetroHealth.lm2$resid)
> myqqnorm(MetroHealth.lm2$resid)
> par(mfrow=c(1,1))  # Return to single graph window.
```

Predicting a Value

The R Code below uses the `predict` function to predict a value of number of doctors given a value of the number of hospitals. The first argument of the function is the model to base the prediction on; the `predict` function is generic, in that it will react appropriately depending on the *class* of the model object given it. Our model object is a "linear model" object, i.e., one created by the `lm` function and `predict` knows what to do with such an object.

The second argument is a data frame giving values of the the explanatory variable(s) that we want to predict response values at. Multiple values are possible, as the second line of R code illustrates.

The first line defines a data frame called `new.data` that contains values of the explanatory variable at which we want to predict response values using the model `MetroHealth.lm1`. An example in the text used a value of 18—hypothetically a hospital in Louisville—for the number of hospitals. We have added a second hypothetical value of 6, just to illustrate that multiple values are possible.

This new data frame becomes the second argument in the second line of code. This line tells the `predict` function to use the model `MetroHealth.lm1` and make two predictions.

The third line of code asks to predict at the same values of 18 and 6, this time using the model `MetroHealth.lm2`. Since this model was for the square-rooted data, we must square the results to get back to the predicted number of doctors. In both cases, the predictions are less than the predictions when using the simple linear model on the untransformed data.

```
new.data <- data.frame(NumHospitals=c(18,6))  # define the data frame to predict at.
predict(MetroHealth.lm1,new=new.data)
       1        2
4691.063 1306.953

predict(MetroHealth.lm2,new=new.data)^2
        1         2
4422.2007  993.6232
```

Adding the Fitted Quadratics—Using R to Draw Curves

We conclude this section on diagnostics by showing how R can create the scatterplot in the original scale with the fitted *quadratic* superimposed. This code suggests a general approach to fitting or drawing curves, which we illustrate with a second example. Note the `main` argument in the `plot` function, which puts a title on the plot.

```
> plot(NumHospitals, NumMDs, main="Doctors vs. Hospitals with Quadratic Fit")
> min(NumHospitals)
[1] 2
> max(NumHospitals)
[1] 32
> xx <- seq(2,32,length=101)
> yy <- (14.033 +2.915*xx)^2
> points(xx,yy,type="l")
```

Explanation of code: First line reproduces the scatterplot. The second and third lines of R code ask for the minimum and maximum values of the x-variable, NumHospitals, which we see to be 2 to 32. Then, we set up a new variable that is a sequence of numbers of length 101, running from 2 to 32. We did not print `xx`, but it equals (2, 2.3, 2.6, 2.9, ..., 31.7, 32). Then, we define the variable `yy` by the quadratic function that untransforms the line we fit to the square-root-transformed data. Finally, the last line of R code uses the `points` function. The `points` function adds the 101 `(xx,yy)` points to the plot BUT using the dictates of that third argument `type='l'`. Without that argument we would have seen 101 new open-circle points added to the scatterplot, but the argument is asking to replace the individual points with a broken line connecting these points. The value of the `type` argument here is a lower case, not an upper case, L, and not the numeral 1. There is nothing magical about 101 in this example, you just want to pick enough points so the curve looks like a smooth curve and not the succession of line segments, which in reality it is.

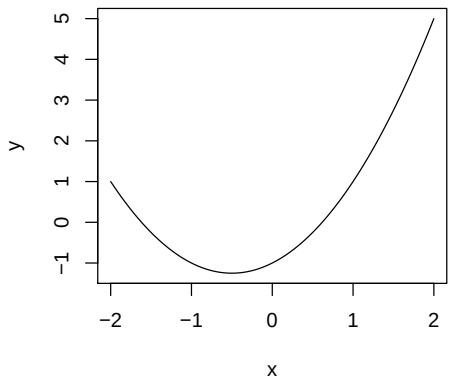
Example 1.A: A mathematical function. We can also graph a mathematical function by itself, not fitted to a scatterplot. Here is an example of code to graph the function $y = x^2 + x - 1$ on the domain $x = [-2, 2]$.

```
x <- seq(-2,2,length=101)
y <- x^2 + x -1

plot(x,y,type="l")

abline(h=0, lty = 2)  # lty=2 makes a dashed line
```

1.5 Outliers and Influential Points

Example 1.9: Butterfly Ballot

We will do more with the topic of outliers and influence when we get to more advanced regression, but the main thing we want to do at this point in our *Companion* is to replicate the scatterplot related to the 2000 U.S. Presidential election where different regression lines are superimposed, depending on whether Palm Beach County is included or not. (This relates to Figure 1.25 in the text.)



Example of a mathematical function: simple quadratic

```
PalmBeach.df <- read.csv(file=file.choose())  # Get the data.

> attach(PalmBeach.df)
> head(PalmBeach.df)
    County Buchanan   Bush
1  ALACHUA      262  34062
2    BAKER       73   5610
3      BAY      248  38637
4 BRADFORD       65   5413
5  BREVARD      570 115185
6  BROWARD      789 177279
> str(PalmBeach.df)
'data.frame':   67 obs. of  3 variables:
 $ County  : Factor w/ 67 levels "ALACHUA","BAKER",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ Buchanan: int  262 73 248 65 570 789 90 182 270 186 ...
 $ Bush    : int  34062 5610 38637 5413 115185 177279 2873 35419 29744 41745 ...


model.with <- lm(Buchanan ~ Bush)

detach()  # To clean up the workspace
```

In the following code, the `!` means negation, so the data from `NoPB.df` has all of the counties except Palm Beach County:

```
# Create a data frame, NoPB.df, removing Palm Beach County.
> attach(PalmBeach.df)
> NoPB.df <- PalmBeach.df[!County=="PALM BEACH",]

dim(NoPB.df)
[1] 66  3

> attach(NoPB.df)
> model.without <- lm(NoPB.df$Buchanan ~ NoPB.df$Bush)

# Now re-create Figure 1.25.
> detach(NoPB.df)
> attach(PalmBeach.df)
> plot(Bush, Buchanan)
> abline(model.with)
> abline(model.without,lty=2)
```

CHAPTER 2

Inference for Simple Linear Regression

Much of the discussion of statistical inference in the regression setting focuses on two tables: the *summary table of coefficients* and *the ANOVA table*.
To obtain the summary table of coefficients, use the `summary` function with the fitted model. The code and output follow. Compare the output to the Minitab output given in the text.

```
> Porsche.df <- read.csv(file=file.choose())  # Get the data.
> attach(Porsche.df)
> Porsche.lm1 <- lm(Price ~ Mileage)
> summary(Porsche.lm1)

Call:
lm(formula = Price ~ Mileage)

Residuals:
     Min       1Q   Median       3Q      Max
-19.3077  -4.0470  -0.3945   3.8374  12.6758

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 71.09045    2.36986   30.00  < 2e-16 ***
Mileage     -0.58940    0.05665  -10.40 3.98e-11 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 7.17 on 28 degrees of freedom
Multiple R-squared: 0.7945,     Adjusted R-squared: 0.7872
F-statistic: 108.3 on 1 and 28 DF,  p-value: 3.982e-11
```

To obtain the ANOVA table, use the `anova` function. The code and output follow, and you can

again compare the results to those given in the text. Notice that R gives us a more precise p-value than does Minitab and that R eliminates the "total" line of the ANOVA table.

```
> anova(Porsche.lm1)
Analysis of Variance Table

Response: Price
          Df Sum Sq Mean Sq F value    Pr(>F)
Mileage    1 5565.7  5565.7  108.25 3.982e-11 ***
Residuals 28 1439.6    51.4
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

To compute confidence intervals for coefficients, use the `confint` function, as in the following examples. Notice that we can get confidence intervals for both (or more generally all) coefficients, or we can specify by number which one(s) we want. We use the `level=.90` argument to obtain 90% confidence intervals. Note that we use a decimal level rather than a percentage, so .90 rather than 90. The default level is, of course, 95%.

```
> confint(Porsche.lm1, level=.90)  # obtain 90% CIs for both coefficients
                   5 %       95 %
(Intercept) 67.0590139 75.1218916
Mileage     -0.6857674 -0.4930345
> confint(Porsche.lm1, 1, level=.90)  # just the intercept
                 5 %     95 %
(Intercept) 67.05901 75.12189
> confint(Porsche.lm1, 2, level=.90)  # just the slope
               5 %       95 %
Mileage -0.6857674 -0.4930345
```

The text explains the *coefficient of determination* or R^2 and shows that it is equal to .795 in this example. One can actually verify that the correlation coefficient, R, can be squared to get this same value.

```
> R <- cor(Price, Mileage)
> R
[1] -0.8913484
> R^2
[1] 0.794502
```

Finally, the section of inference for simple linear regression discusses "intervals for prediction," namely the confidence interval and the prediction interval. One obtains these using the R `predict` function along with the `int` argument. We give below the code and output for the example of a Porsche with 50 thousand miles.

```
# First the 95% Confidence Interval:
new.data <- data.frame(Mileage = 50)
predict(Porsche.lm1,new.data, int="confidence")

# Here is the output:
       fit      lwr      upr
1 41.62041 38.41535 44.82546

# Next the 95% Prediction Interval:
predict(Porsche.lm1,new.data, int="prediction")
       fit      lwr     upr
1 41.62041 26.58711 56.6537
```

Notice the agreement to the textbook.

One can again use the `level` argument to obtain something other than 95% confidence levels. Here is an example of a 90% prediction interval. As expected, it is narrower than the 95%.

```
predict(Porsche.lm1,new.data, int="prediction", level=.90)
       fit      lwr      upr
1 41.62041 29.13577 54.10504
```

Graphing Prediction and Confidence Intervals

Figure 2.2 in the text shows a scatterplot of Mileage versus Price for the Porsche data, along with the fitted regression line and with added `confidence bands` and `prediction bands` that graphically present the 95% confidence and prediction interval values at each level of the explanatory variable, Price.

Below, we give R code for reproducing these graphs. Rather than giving the code for this particular example only, we will instead give code to define a new function called `predict.plots` that can be entered and saved into R and used with future simple linear regression models. Two nuances require explanation in this code. First, we use a `range` function call in the `plot` function call. The result is to set the `ylim` argument—which sets a range on the y-axis—to be big enough to encompass all points of the plot and the bands, so that none of the picture is out of range. Then, we introduce the `matpoints` function, which plots the vector `x` against all columns of the `CI` or `PI` matrix, exactly what we require here. (We could use a sequence of regular `points` function calls, but our way is more expedient.) To create Figure 2.2 we then type `predict.plots(Mileage,Price)`.

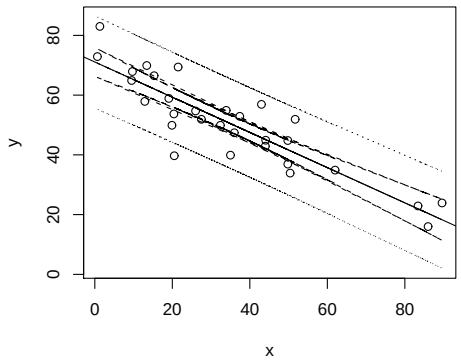


Figure 2.2 (page 78): Confidence bands and prediction bands for Porsche prices

```
predict.plots <- function(x,y, conf.level=.95) {
          # x = explanatory variable;
          # y = response variable.
          model <- lm(y~x)
          new <- seq(min(x),max(x),length=101)
          CI <- predict(model, list(x = new), int="confidence", level=conf.level)
          PI <- predict(model, list(x = new), int="prediction", level=conf.level)
          plot(x,y,ylim=range(y,PI[,3]))
          abline(model)  # to obtain solid regression line
          points(new,CI[,2],type="l",col=1,lty=2)
          points(new,CI[,3],type="l",col=1,lty=2)
          points(new,PI[,2],type="l",col=1,lty=3)
          points(new,PI[,3],type="l",col=1,lty=3)
}
```

CHAPTER 3

Multiple Regression

Our R companion now moves on to multiple regression and other regression topics beyond simple linear. These regression topics introduce us to several new ideas, for which we will want to learn the appropriate R syntax. In this section, we will describe how to use R to help with:

- Indicator variables,
- Adjusted R^2,
- Interaction terms,
- Coded scatterplots,
- Matrix plots,
- Correlation matrices,
- Variance inflaction factors,
- Polynomial regression, and
- Nested F-tests.

Conceptually, of course, there is more to multiple linear regression than this list implies, but much of the R we learned for simple linear regression works the same way in the multiple-predictor context. For example, the `lm` function still fits the model, `summary` and `anova` still provide important tabular output, the model object still embodies important components such as fitted values or residuals, and the `predict` function still helps with prediction and its related confidence or prediction intervals. We will proceed to explicate the R items in the list above by using R to reproduce solutions to several of the multiple regression examples from the text.

Example 3.1: NFL Winning Percentage

```
> NFLStandings2007.df <- read.csv(file=file.choose()) # Get the data!
> attach(NFLStanding2007.df)
```

```
> plot(PointsFor,WinPct)
> plot(PointsAgainst,WinPct)
```

These two commands produce Figures 3.1(a) and (b) without the regression lines being included.

3.1 Multiple Linear Regression Model

Example 3.2: NFL Winning Percentage (continued)

The commands `NLF.lm1 <- lm(WinPct ~ PointsFor + PointsAgainst)` and `summary(NFL.lm1)` produce the fitted regression model as shown in the book, including the t-ratio of -5.55 for testing that there is no linear relationship between WinPCT and PointsAgainst in the model that also includes PointsFor.

```
> NLF.lm1 <- lm(WinPct ~ PointsFor + PointsAgainst)
> summary(NFL.lm1)

Call:
lm(formula = WinPct ~ PointsFor + PointsAgainst)

Residuals:
     Min       1Q   Median       3Q      Max
-0.15857 -0.05318 -0.01259  0.07360  0.12962

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.4172230  0.1394480   2.992  0.00561 **
PointsFor      0.0017662  0.0001870   9.445 2.37e-10 ***
PointsAgainst -0.0015268  0.0002751  -5.551 5.50e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.07298 on 29 degrees of freedom
Multiple R-squared: 0.8844,Adjusted R-squared: 0.8764
F-statistic: 110.9 on 2 and 29 DF,  p-value: 2.598e-14
```

3.2 Assessing a Multiple Regression Model

Example 3.5: NFL Winning Percentage (continued)

From `summary(NFL.lm1)` we see that R^2 is 0.8844, which R calls "Multiple R-squared." We could also have found this by first having R compute the fitted values from the regression and then finding the correlation of those values with the response variable (WinPct) and squaring.

```
> fits <- NFL.lm1$fitted
> cor(WinPct,fits)^2
```

Example 3.6: NFL Winning Percentage (continued)

The adjusted R^2 is given in the output from the `summary(NFL.lm1)` command as 0.8764.

Example 3.7: NFL Winning Percentage (continued)

To make predictions as well as confidence intervals and prediction intervals for those predictions we first create a new data frame with numbers for each of the predictor variables; then we use the `predict` command as in Chapter 2.

```
> newdata <- data.frame(PointsFor=400,PointsAgainst=350)
> predict.lm(NFL.lm1,newdata, int="confidence")
        fit       lwr       upr
1 0.5893095 0.5555032 0.6231158

> predict.lm(NFL.lm1,newdata, int="prediction")
        fit       lwr       upr
1 0.5893095 0.4362648 0.7423541
```

3.3 Comparing Two Regression Lines

Example 3.9: Growth Rate of Kids

We begin with the example of comparing rates of weight gain for boys and girls using a sample of 198 children and observing them from about age 8 to age 18. (Ages are given in months in the dataset.) Here we use a model with the sex of the child as a binary predictor in a model that also includes the child's age. Recall that by considering an interaction term we can model any differential rates of weight gain between sexes.

```
> Kids198.df <- read.csv(file=file.choose()) # Get the data!
> attach(Kids198.df)

> plot(Age,Weight,pch=16-15*Sex) # Sex=0 (boys) gives plot symbol 16;
> legend(locator(1),c("boys","girls"),pch=c(16,1)) # Sex=1 (girls) gives symbol 1

> abline(lm(Weight[Sex==0] ~ Age[Sex==0]))  # Boys are coded as 0.
> abline(lm(Weight[Sex==1] ~ Age[Sex==1]),lty=2)  # Girls are coded as 1.
```

Explanation: After attaching the data frame Kids198.df, we reproduce the scatterplot. We use the `legend` function to add the legend. The `locator(1)` argument suspends command line mode,

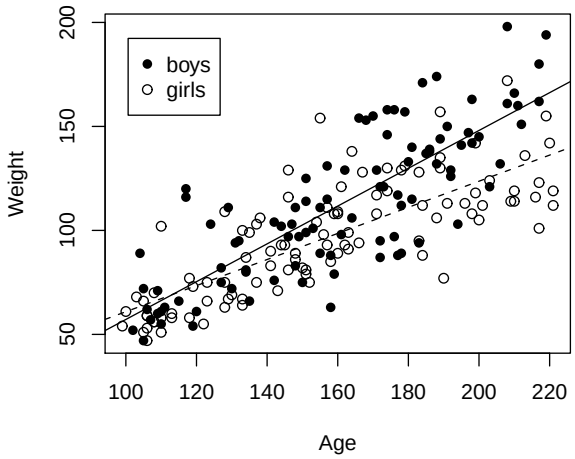


Figure 3.9 (page 114): Compare regression lines by sex

asking the user to select where in the plot to add the legend. The user drags the cursor over into the graph region and clicks at the position where he or she wants the upper left-hand corner of the legend to be. The second argument of `legend` gives the point labels (boys and girls; note the quote marks), and the `pch` argument gives the symbols relating to labels; here `pch=16` represents a solid circle symbol and `pch=1` an open circle.

Finally, we add the two regression lines, solid (default line style) for boys and dashed (`lty=2`) for girls.

We now fit the linear model of `Weight` on `Age` and `Sex` with an interaction term, as shown in the text, save for using `Sex` as the binary for sex rather than creating the indicator `IGirl`. In R we designate the interaction term by `Age:Sex` which tells R to include the point-wise product of the two numeric variables `Age` and `Sex`. A more cryptic notational option exists; the model statement below could be replaced by `lm(Weight ~ Age*Sex`, which automatically includes the interaction term along with lower-order terms (in this case, the linear terms and the intercept).

```
> WeightAge.lm1 <- lm(Weight ~ Age + Sex + Age:Sex)
> summary(WeightAge.lm1)

Call:
lm(formula = Weight ~ Age + Sex + Age:Sex)
```

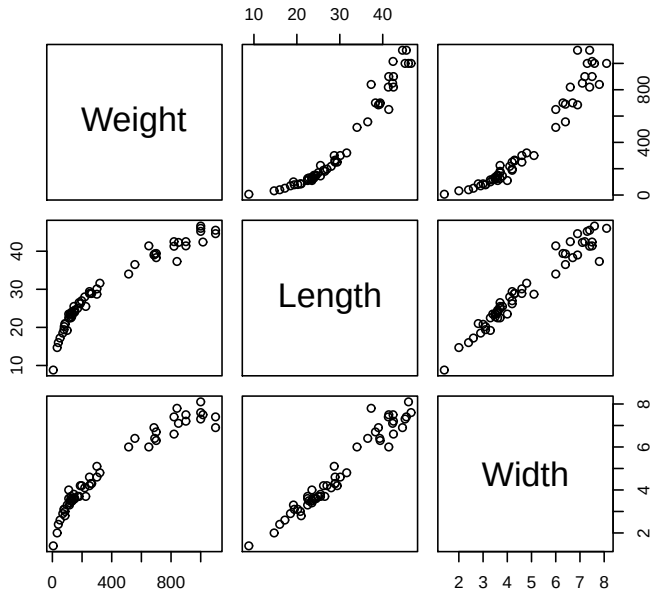
```
Residuals:
    Min      1Q  Median      3Q     Max
-46.884 -12.055  -2.782  10.185  58.581

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -33.69254   10.00727  -3.367 0.000917 ***
Age           0.90871    0.06106  14.882  < 2e-16 ***
Sex          31.85057   13.24269   2.405 0.017106 *
Age:Sex      -0.28122    0.08164  -3.445 0.000700 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 19.19 on 194 degrees of freedom
Multiple R-squared: 0.6683,     Adjusted R-squared: 0.6631
F-statistic: 130.3 on 3 and 194 DF,  p-value: < 2.2e-16
```

3.4 New Predictors from Old

Example 3.10: Perch Weights

We give below the R code for several pieces of the Perch Weight example in the text. The `pairs` function is new and produces a *matrix* plot, that is, a plot of all possible pairwise scatterplots of the variables given in the model statement (in the order given). We might have typed `pairs(Perch.df)` except that would have included the `Obs` variable in the plot, which is not useful. We use the `cor` function, which computes correlation coefficients between pairs of variables. By entering the argument `cbind(Weight,Length,Width)` we have given the function a matrix with 3 columns, causing the calculation of all pairwise correlation coefficients. Note the use of the `round` function, to 3 decimal places, to make the output more appealing.

```
> Perch.df <- read.csv(file=file.choose()) # Get the data!
> dim(Perch.df)
[1] 56  4
> names(Perch.df)
[1] "Obs"    "Weight" "Length" "Width"

> attach(Perch.df)
> pairs(~ Weight + Length + Width) # produces a matrix plot of the 3 variables
> round(cor(cbind(Weight,Length,Width)),3) # pairwise correlations, rounded.
```



Matrix plot for Perch Weights

```
       Weight Length Width
Weight  1.000  0.960 0.964
Length  0.960  1.000 0.975
Width   0.964  0.975 1.000
```

Below is code for the the first two models the text discusses for these data. The first model—`Perch.lm1`—fits only linear terms for `Length` and `Width`, while the second model—`Perch.lm2`—adds an interaction term.

```
> Perch.lm1 <- lm(Weight ~ Length + Width)
> summary(Perch.lm1)

Call:
lm(formula = Weight ~ Length + Width)

Residuals:
    Min      1Q  Median      3Q     Max
-113.86  -59.02  -23.29   30.93  299.85
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -578.758     43.667 -13.254  < 2e-16 ***
Length        14.307      5.659   2.528 0.014475 *
Width        113.500     30.265   3.750 0.000439 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 88.68 on 53 degrees of freedom
Multiple R-squared: 0.9373,     Adjusted R-squared: 0.9349
F-statistic: 396.1 on 2 and 53 DF,  p-value: < 2.2e-16

> Perch.lm2 <- lm(Weight ~ Length + Width + Length:Width)
> summary(Perch.lm2)

Call:
lm(formula = Weight ~ Length + Width + Length:Width)

Residuals:
     Min       1Q   Median       3Q      Max
-140.106  -12.226    1.230    8.489  181.408

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  113.9349    58.7844   1.938    0.058 .
Length        -3.4827     3.1521  -1.105    0.274
Width        -94.6309    22.2954  -4.244 9.06e-05 ***
Length:Width   5.2412     0.4131  12.687  < 2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 44.24 on 52 degrees of freedom
Multiple R-squared: 0.9847,     Adjusted R-squared: 0.9838
F-statistic:  1115 on 3 and 52 DF,  p-value: < 2.2e-16
```

Recall that both model objects are lists of length 12, components of which include constituents for diagnostic plots. For example:

```
> names(Perch.lm1)
 [1] "coefficients"  "residuals"  "effects"  "rank" "fitted.values" "assign"
 [7] "qr"    "df.residual"  "xlevels"   "call"   "terms"         "model"
```

This is a reasonable time to introduce a feature of R's that gives easy access to regression assessment plots. If you simply enter the single command:

```
plot(Perch.lm1)
```

R will give you a succession of 4 plots involving residuals, including two we have discussed in both the *Companion* and the text: residuals versus fitted values and a normal plot of the residuals.

This command is another instance of R being flexible in implementation of a function—here the `plot` function—that recognizes the input object as being a regression object and reacts appropriately.

The perch data also provide opportunity to illustrate two other important regression topics: the *Nested F-test* and *polynomial models*. Below we fit a third model to these data, one also given in the text, the so-called "second-order model." Notice the agreement of the output with that given in the text.

Notational aside We move now to Example 3.12. The model statement for `Perch.lm3` contains an operator we have not seen yet in the *Companion*, the `I` operator. As mentioned above, the model notation in R uses the `*` symbol in a special way: not as the multiplication operation for two variables, but as a way of signaling the inclusion of a model with interaction term along with linear terms (that is, the terms marginal to the interaction term). While R admits a special notation—`Length:Width`—to denote the product of two numeric variables, it also provides the `I` operator to "protect" a mathematical operator, such as `*` that would otherwise be interpreted non-mathematically on the right-hand side of a model statement. Thus, an optional way to write our second model above is `Perch.lm2 ~ Length + Width + I(Length*Width)`.

Without the use of `I` in the model statement (below) for `Perch.lm3`, the term for $\texttt{Length}^2$ would be interpreted as the interaction of Length with itself, which would reduce to just Length; likewise for $\texttt{Width}^2$. This would give a model equivalent to Perch.lm2. (Try it and see!) But with the `I()` notation, R interprets the term in a mathematical sense, thus fitting the square of length, as desired in this case.

```
> Perch.lm3 <- lm(Weight ~ Length + Width + I(Length^2) +
+ I(Width^2) + Length:Width)
> summary(Perch.lm3)

Call:
lm(formula = Weight ~ Length + Width + I(Length^2) + I(Width^2) +
    Length:Width)

Residuals:
     Min       1Q   Median       3Q      Max
-117.175  -11.904    2.822   11.556  157.596
```

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  156.3486    61.4152   2.546   0.0140 *
Length       -25.0007    14.2729  -1.752   0.0860 .
Width         20.9772    82.5877   0.254   0.8005
I(Length^2)    1.5719     0.7244   2.170   0.0348 *
I(Width^2)    34.4058    18.7455   1.835   0.0724 .
Length:Width  -9.7763     7.1455  -1.368   0.1774
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 43.13 on 50 degrees of freedom
Multiple R-squared: 0.986,      Adjusted R-squared: 0.9846
F-statistic: 704.6 on 5 and 50 DF,  p-value: < 2.2e-16
```

One can use this example as a guide for fitting polynomial regression models or, more generally, models with some polynomial terms.

3.5 Correlated Predictors and Variance-Inflation Factors

Example 3.15: Diamond Prices

The R code and output below indicate fitting the model, given in the text, for predicting the price of a diamond from its weight (in carats) and the depth of the cut. This code also shows how to obtain variance-inflation factors in R. The `vif` function resides in an R library called car; hence the line `library(car)`.

```
> Diamonds.df <- read.csv(file=file.choose())
         #obtain the data set; R allows you to search for it.
> attach(Diamonds.df)
> Diamond.lm <- lm(TotalPrice ~ Carat + I(Carat^2) + Depth)
> summary(Diamond.lm)

Call:
lm(formula = TotalPrice ~ Carat + I(Carat^2) + Depth)

Residuals:
      Min        1Q    Median        3Q       Max
-11166.72   -713.88    -52.67    563.94  11263.69
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  6343.09    1436.49   4.416 1.35e-05 ***
Carat        2950.04     736.11   4.008 7.51e-05 ***
I(Carat^2)   4430.36     254.65  17.398  < 2e-16 ***
Depth        -114.08      22.66  -5.034 7.74e-07 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 2056 on 347 degrees of freedom
Multiple R-squared: 0.9308,     Adjusted R-squared: 0.9302
F-statistic:  1555 on 3 and 347 DF,  p-value: < 2.2e-16

> library(car)  # Need the car library for the vif function

Attaching package: 'car'

The following object(s) are masked _by_ '.GlobalEnv':

    subsets

> vif(Diamond.lm)
     Carat I(Carat^2)      Depth
 10.942252  10.718736   1.117426
```

3.6 The Nested F-test and Taking Time for an "R Moment"

The text illustrates the Nested F-test using the models `Perch.lm2` and `Perch.lm3`.

We take `Perch.lm3`, the second-order model, as our "full model" and compare it to `Perch.lm2`, the model with linear terms and the interaction term, which becomes our "reduced" model. Recall the formula for the F-statistic (our test statistic) is

$$F = \frac{(SSModel_{full} - SSModel_{reduced})/\#\text{ predictors tested}}{SSE_{full}/(n-k-1)}$$

To obtain the model sums of squares we can use the `anova` function as the code below illustrates.

```
REDUCED MODEL:
> anova(Perch.lm2)
Analysis of Variance Table
```

```
Response: Weight
             Df  Sum Sq Mean Sq  F value    Pr(>F)
Length        1 6118739 6118739 3126.571 < 2.2e-16 ***
Width         1  110593  110593   56.511 7.416e-10 ***
Length:Width  1  314997  314997  160.958 < 2.2e-16 ***
Residuals    52  101765    1957
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

# Notice that we can print out just the set of sums of squares
# from the ANOVA table using the [[2]] notation.

> anova(Perch.lm2)[[2]]
[1] 6118739.4  110592.9  314997.3  101764.7

FULL MODEL:
> anova(Perch.lm3)
Analysis of Variance Table

Response: Weight
             Df  Sum Sq Mean Sq   F value    Pr(>F)
Length        1 6118739 6118739 3289.6413 < 2.2e-16 ***
Width         1  110593  110593   59.4585 4.667e-10 ***
I(Length^2)   1  314899  314899  169.3002 < 2.2e-16 ***
I(Width^2)    1    5381    5381    2.8932   0.09517 .
Length:Width  1    3482    3482    1.8719   0.17737
Residuals    50   93000    1860
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

So $SSModel_{reduced} = 6118739+110593+314997 = 6544329$ and $SSModel_{full} = 6118739+110593+314899+5381+3482 = 6553094$, and the difference is $8765 = 6118739 - 6544329$. One can equivalently get this value as the difference in the Sum of Squares for Residuals, $101765 - 93000 = 8765$. We note that this value of 8765 is a rounded value (the text actually has 8764 for its rounded value).

We end our discussion of nested F-test with an "R moment" by which we mean an opportunity to be reminded of the power and wonder of R. The above discussion illustrates that with what we know to this point we can calculate the F-test, but as is so often the case with R, there is another—need we say "easier"?—way, using the same `anova` function used above. In general, the command line `anova(reducedmodel, fullmodel)` does the calculation in one fell swoop. It would be a sign of gaining R experience to find this unsurprising. R often, if not usually, contains a function that will perform an operation that is frequently encountered in the statistical world. Below is the R code for the example at hand.

```
> anova(Perch.lm2, Perch.lm3)
Analysis of Variance Table

Model 1: Weight ~ Length + Width + Length:Width
Model 2: Weight ~ Length + Width + I(Length^2) + I(Width^2) + Length:Width
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     52 101765
2     50  93000  2    8764.6 2.3561 0.1052
```

CHAPTER 4

Additional Regression Topics

We now illustrate R code required for the various additional regression topics from Chapter 4 of the text by using many of the same examples given in that chapter.

4.1 Added Variable Plots

Example 4.1: House Prices

First, we create with R the added variable plots. We begin with the code for getting the data and fitting the linear model of *Price* on the two predictors of *Lot* and *Size*, which are the lot size and the house size. We also confirm that the correlation between the two predictors is 0.716.

```
> Houses.df <- read.csv(file=file.choose()) # Get the data!
> names(Houses.df)
[1] "Price" "Size"  "Lot"

> attach(Houses.df)
> Houses.lm1 <- lm(Price ~ Lot + Size)
> summary(Houses.lm1)

Residuals:
   Min     1Q Median     3Q    Max
-79532 -28464   3713  21450  73507

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 34121.649  29716.458   1.148   0.2668
Lot             5.657      3.075   1.839   0.0834 .
Size           23.232     17.700   1.313   0.2068
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
Residual standard error: 47400 on 17 degrees of freedom
Multiple R-squared: 0.5571,     Adjusted R-squared: 0.505
F-statistic: 10.69 on 2 and 17 DF,  p-value: 0.000985

> cor(Price, Lot)
[1] 0.7157072
```

Next, we create the added variable plot for *Size* after *Lot*. This plot shows the relationship between the unexplained variability in *Price* fit to *Lot* (`Error1` below) to the variability in *Size* that cannot be explained by *Lot* (`Error2` below).

```
> Price.Lot.lm <- lm(Price ~ Lot)
> Error1 <- Price.Lot.lm$resid
> Size.Lot.lm <- lm(Size ~ Lot)
> Error2 <- Size.Lot.lm$resid
> Error1.Error2.lm <- lm(Error1 ~ Error2)
> Error1.Error2.lm

Call:
lm(formula = Error1 ~ Error2)

Coefficients:
(Intercept)       Error2
  1.597e-12    2.323e+01

> plot(Error2, Error1, main="Added Variable Plot")
> abline(Error1.Error2.lm)
```

Alternatively—as you might by now have anticipated—there is an added variable function in R that does the task in a single function call. The function is in the `car` library and is called `avPlots`. We illustrate it below, producing the same plot we just constructed. The output gives both added variable plots: `Lot` after `Size` and then `Size` after `Lot`. Note the `col=1` argument, which changes the color to black, where the default is red.

```
> library(car)
> avPlots(Houses.lm1, col=1)
```

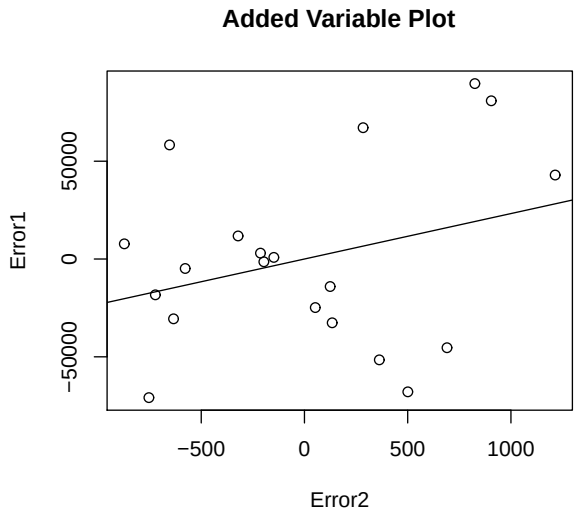
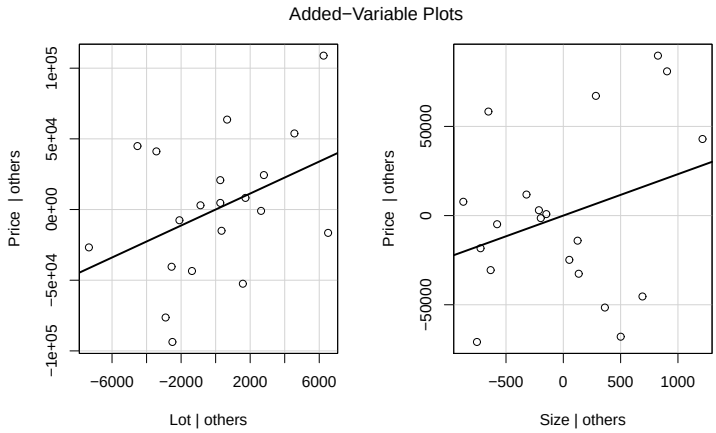


Figure 4.3 (page 167): Added Variable Plot for adding Size to Lot when predicting Price



Added Variable Plot for Houses data using `avPlots`

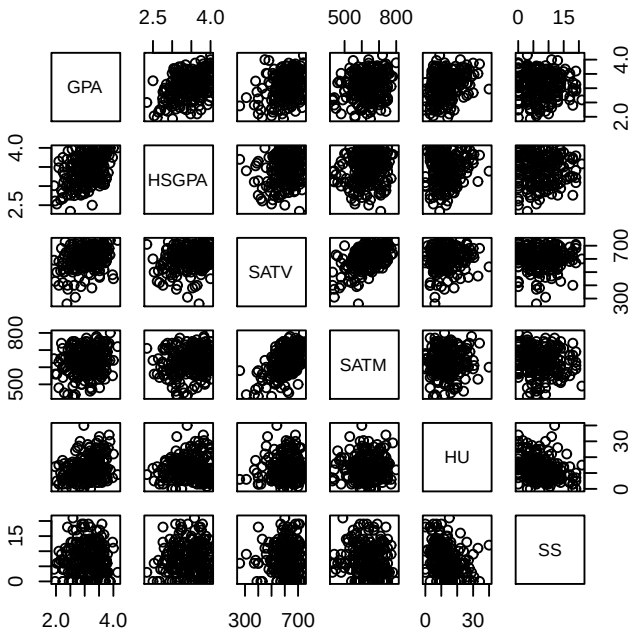
4.2 Techniques for Choosing Predictors

Example 4.2: The GPA Data

We now illustrate R code for using automated procedures for model selection, using the same dataset the text uses. After getting the data and attaching it, we ask for variable names. To replicate the matrix scatterplot in the text, we note that the variables we want in the plot—GPA, HSGPA, SATV, SATM, HU, and SS—are variables 1, 2, 3, 4, 6, and 7 among the 10 variables in

the dataset. (We don't want the binary variables in the matrix plot.) Therefore, the call to `pairs` subsets out these columns from the GPA data frame.

```
> GPA.df <- read.csv(file=file.choose())
> attach(GPA.df)
> names(GPA.df)
 [1] "GPA"          "HSGPA"        "SATV"         "SATM"         "Male"
 [6] "HU"           "SS"           "FirstGen"     "White"        "CollegeBound"
> pairs(GPA.df[,c(1,2,3,4,6,7)])
```



Scatterplot matrix for first year GPA data

We compare GPAs for the values of our 4 binary predictors using boxplots. We use `par` to set up the graphics window as a 2-by-2 array of 4 graph panels. We then use 4 `boxplot` function calls, using the `main` argument to add a title and using the `horizontal` argument to align the boxplots horizontally, the default being vertically. Note that R orders the plots differently from the order in the textbook, which come from Minitab.

```
> par(mfrow=c(2,2)) # Set graphics window to a 2-by-2 array of 4 panels.
> boxplot(GPA ~ Male, main="1=male; 0=female",horizontal=TRUE)
> boxplot(GPA ~ FirstGen, main="1=first generation; 0=not", horizontal=TRUE)
```

```
> boxplot(GPA ~ White, main="1=white; 0=others", horizontal=TRUE)
> boxplot(GPA ~ CollegeBound, main="1=more than half to college", horizontal=TRUE)
> par(mfrow=c(1,1)) # set graphics window back to one panel
```

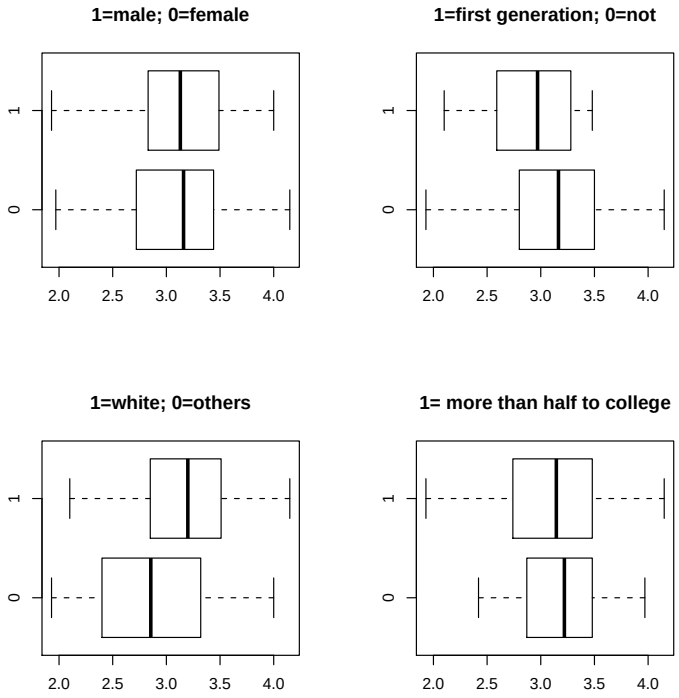


Figure 4.5 (page 170): GPA versus categorical predictors

To obtain a *best subsets* analysis, we use a function in the `leaps` library called `regsubsets`. We give the appropriate code and output below. Let's explain the code, line by line. First, we fit the same model first fit in the textbook. We suppress the summary table in the second line, but you can check that it replicates exactly that found in the book. In the third line we fit the model that includes all predictors, including the `CollegeBound` indicator, which is highly non-significant. Notice here we use the compact option for specifying the model: `GPA` $\sim$ `.` tells R to regress GPA on all predictors. If you look at the summary table, you will notice it closely resembles the first summary because of the lack of significance of `CollegeBound`.

We then call up the `leaps` library, that contains the critical function: `regsubsets`. We then define the matrix `X`, whose columns form the set of predictors—in this case all columns of `GPA.df` except that first column; we then define `Y` to be our response variable, that is, the first column of our data frame. In the eighth line we use the `regsubsets` function that extracts a best subsets analysis of our predictors with our response. The `nvmax` argument constrains models to have 8 or fewer

variables; the `nbest` argument constrains output to report the 2 best models for each number of variables. We have assigned the summary of the `regsubsets` call to a variable called `out` and we then summarize that object, which is a list of length 8, the pieces of which will become important for what follows.

The `which` component is the guts of the summary table; each line corresponds to a model with 1 to 8 variables. There are 17 lines because there is only one model with all 8 predictors (obviously). We place these 17 lines into an object called `Subsets`. We see these 17 lines in the textbook example, but we won't get information such as R^2 or Mallow's C_p unless we access other pieces from `out`.

Subsequent lines pick off the R^2, adjusted R^2, and Mallow's C_p quantities (which we round for more graceful presentation), and the final line of code makes a summary table that replicates the information that Minitab gave us in the textbook example.

```
> full <- lm(GPA ~ HSGPA + SATV + Male + HU + SS + White, data=GPA.df)
> summary(full) # we suppress the output
> full <- lm(GPA ~ ., data=GPA.df) # notice the short way to specify the model
> summary(full)  # we suppress the output, but it agrees with Minitab's
> library(leaps) # leaps library contains the regsubsets function
> out=summary(regsubsets(GPA~.,nbest=2,data=GPA.df))
> Subsets <- out$which
> R2 <- round(100*out$rsq,1)
> R2adj <- round(100*out$adjr2,1)
> Cp <- round(out$cp,1)
> cbind(as.data.frame(Subsets),R2,R2adj,Cp)

(We have suppressed the output, which agrees with the Minitab
output in the text.)
```

To obtain a backward elimination calculation we can use the `step` function. In the code below, notice that we first must fit the full model—that is the model that uses all 9 predictors—and this full model becomes the input to the `step` function.

Below the code comes the output. Because `step` is using the AIC criterion to eliminate variables, rather than p-values, the results differ from the Minitab results the text presents. The final model given by R includes predictors `HSGPA`, `SATV`, `HU`, `SS`, and `White`, which is the penultimate model Minitab gives, before eliminating `SS` on its final iteration.

One can also use `step` to perform forward selection, by setting the `direction` argument to `forward`. Try `help(step)` to get more explanation of this function.

```
> attach(GPA.df)
> full <- lm(GPA ~ HSGPA+SATV+SATM+Male+HU+SS+FirstGen+White+CollegeBound)
```

```
> full <- lm(GPA ~ ., data= GPA.df)
            # this is an alternative to the first 2 lines

> step(full)

Start:  AIC=-410.16
GPA ~ HSGPA + SATV + SATM + Male + HU + SS + FirstGen + White +
    CollegeBound

               Df Sum of Sq    RSS     AIC
- SATM          1    0.0053 30.724 -412.12
- CollegeBound  1    0.0067 30.726 -412.11
- FirstGen      1    0.1031 30.822 -411.42
- Male          1    0.1052 30.824 -411.41
- SS            1    0.2556 30.975 -410.34
<none>                      30.719 -410.16
- SATV          1    0.3309 31.050 -409.81
- White         1    1.1545 31.873 -404.08
- HU            1    2.4409 33.160 -395.41
- HSGPA         1    6.4345 37.154 -370.51

Step:  AIC=-412.12
GPA ~ HSGPA + SATV + Male + HU + SS + FirstGen + White + CollegeBound

               Df Sum of Sq    RSS     AIC
- CollegeBound  1    0.0065 30.731 -414.07
- FirstGen      1    0.1072 30.832 -413.36
- Male          1    0.1421 30.866 -413.11
- SS            1    0.2503 30.975 -412.34
<none>                      30.724 -412.12
- SATV          1    0.4532 31.178 -410.91
- White         1    1.1778 31.902 -405.88
- HU            1    2.4567 33.181 -397.27
- HSGPA         1    6.5762 37.301 -371.64

Step:  AIC=-414.07
GPA ~ HSGPA + SATV + Male + HU + SS + FirstGen + White
```

```
           Df Sum of Sq    RSS     AIC
- FirstGen  1    0.1129 30.844 -415.27
- Male      1    0.1469 30.878 -415.03
- SS        1    0.2470 30.978 -414.32
<none>                  30.731 -414.07
- SATV      1    0.4677 31.199 -412.77
- White     1    1.1713 31.902 -407.88
- HU        1    2.4506 33.181 -399.27
- HSGPA     1    6.7560 37.487 -372.55

Step:  AIC=-415.27
GPA ~ HSGPA + SATV + Male + HU + SS + White

        Df Sum of Sq    RSS     AIC
- Male   1    0.1534 30.997 -416.18
- SS     1    0.2815 31.125 -415.28
<none>               30.844 -415.27
- SATV   1    0.5898 31.434 -413.12
- White  1    1.2934 32.137 -408.27
- HU     1    2.8154 33.659 -398.14
- HSGPA  1    6.6441 37.488 -374.55

Step:  AIC=-416.18
GPA ~ HSGPA + SATV + HU + SS + White

        Df Sum of Sq    RSS     AIC
<none>               30.997 -416.18
- SS     1    0.2951 31.292 -416.11
- SATV   1    0.7005 31.698 -413.29
- White  1    1.3133 32.310 -409.10
- HU     1    2.7987 33.796 -399.25
- HSGPA  1    6.4968 37.494 -376.51

Call:
lm(formula = GPA ~ HSGPA + SATV + HU + SS + White, data = GPA.df)

Coefficients:
(Intercept)        HSGPA         SATV           HU           SS        White
  0.5684876    0.4739983    0.0007481    0.0167447    0.0077474    0.2060408
```

4.3 Identifying Unusual Points in Regression

Example 4.5: More Butterfly Ballots

The textbook introduces these quantities for identifying unusual points in a regression analysis:

- Leverage,
- Standardized and studentized residuals, and
- Cook's distance.

All of these quantities are available through the `ls.diag` function, the output of which can be used to identify *moderately unusual* or *very unusual* points according to the summary table in the text, which we reproduce below. Recall that k is the number of predictors and n is the number of cases.

Statistic	Moderately unusual	Very unusual
Leverage, h_i	above $2(k+1)/n$	above $3(k+1)/n$
Standardized residual	beyond ± 2	beyond ± 3
Studentized residual	beyond ± 2	beyond ± 3
Cook's D	above 0.5	above 1.0

For this example, we return to the `PalmBeach.df` data frame and form the model that regresses `Buchanan` on `Bush`; we name the model `PalmBeach.lm`. We obtain a collection of regression diagnostics using the code below. The result of the call to `ls.diag` is a list of 8 objects that includes the hat matrix entries (i.e., the h_i values), standardized residuals, studentized residuals, and Cook's distance, labeled respectively `hat`, `std.res`, `stud.res`, and `cooks` in the output. (*Note:* You can get a fuller explanation of all components of the list by typing `help(ls.diag)`.)

```
> attach(PalmBeach.df)
> PalmBeach.lm <- lm(Buchanan ~ Bush)
> PalmBeach.diag <- ls.diag(PalmBeach.lm)
> summary(PalmBeach.diag)
             Length Class  Mode
std.dev       1     -none- numeric
hat          67     -none- numeric
std.res      67     -none- numeric
stud.res     67     -none- numeric
cooks        67     -none- numeric
dfits        67     -none- numeric
correlation   4     -none- numeric
std.err       2     -none- numeric
cov.scaled    4     -none- numeric
cov.unscaled  4     -none- numeric
```

To replicate the textbook's output that identifies the counties that exceed the threshold of $h_i > 6/n$ you can type in this code:

```
> PalmBeach.df[PalmBeach.diag$hat > 6/67,] # We want all columns that satisfy
                                           # the row condition, thus we use the
                                           # wild card convention of blank space
                                           # after the comma.

         County Buchanan   Bush
6       BROWARD      789 177279
13         DADE      561 289456
29 HILLSBOROUGH      836 176967
52     PINELLAS     1010 184312
```

To obtain those cases that exceed either the $h_i > 6/n$ threshold or the threshold of the standardized residuals exceeding ± 2, we can use the following code. Notice the use of the | symbol to denote the logical "or."

```
# in the first two lines, we take the set of all subscripts---1, 2, ..., 67---
# and select out only those subscripts we want to look at.  The 67 subscripts
# correspond to the 67 Florida counties.
> id <- (1:67)[PalmBeach.diag$hat > 6/67 | abs(PalmBeach.diag$std.res) > 2]
> id
[1]  6 13 29 50 52

> results <- cbind(PalmBeach.df[id,],PalmBeach.lm$fit[id],
+ PalmBeach.lm$resid[id],PalmBeach.diag$std.res[id])
> names(results) <- c("County","Buchanan","Bush","Fits",
+ "Resids","Stand. Resids")
> results
         County Buchanan   Bush      Fits     Resids Stand. Resids
6       BROWARD      789 177279  916.9402 -127.94023    -0.3807500
13         DADE      561 289456 1468.4953 -907.49526    -3.0591800
29 HILLSBOROUGH      836 176967  915.4062  -79.40618    -0.2362617
50   PALM BEACH     3407 152846  796.8074 2610.19263     7.6510719
52     PINELLAS     1010 184312  951.5203   58.47972     0.1749128
```

Finally, here is code to replicate Table 4.1 of unusual cases in the perch weights multiple regression example. In this example, `Perch.df` is the perch weight data frame, with variables `Length`, `Width`, and `Weight`, and `Perch.lm2` is the regression model of `Weight` on `Length`, `Width`, and the interaction term `Length:Width`.

```
> Perch.diag <- ls.diag(Perch.lm2)
> id <- (1:56)[Perch.diag$hat > 8/56 | abs(Perch.diag$std.res) > 2]
> id
[1]  1  2 40 50 52 55 56

> StRes <- round(Perch.diag$std.res[id],2)
> Hats <- round(Perch.diag$hat[id],3)
> Cooks <- round(Perch.diag$cooks[id],3)

> cbind(Perch.df[id,c(3,4,2)],StRes,Hats,Cooks)
   Length Width Weight StRes  Hats Cooks
1     8.8   1.4    5.9 -0.28 0.431 0.015
2    14.7   2.0   32.0  0.11 0.153 0.001
40   37.3   7.8  840.0  1.96 0.363 0.547
50   42.4   7.5 1015.0  2.16 0.078 0.098
52   44.6   6.9 1100.0  4.37 0.121 0.655
55   46.0   8.1 1000.0 -3.44 0.151 0.525
56   46.6   7.6 1000.0 -2.17 0.143 0.196
```

4.4 Coding Categorical Predictors

The file **ThreeCars** includes indicator variables `Porsche`, `Jaguar`, and `BMW` already created. If this were not true we would use the R command

```
> Porsche <- as.numeric(CarType=="Porsche")
```

to create the indicator variable for Porsche and likewise for Jaguar and BMW.

If we wanted to produce the first of the predictions and intervals found in the text, we would first fit the regression model with the command

```
> threecars.lm1=lm(Price~Mileage+Porsche+Jaguar)
```

and then create a (tiny) data frame for a Porsche with 50,000 miles using

```
> newx <- data.frame(Mileage=50,Porsche=1,Jaguar=0,BMW=0)
```

Finally, we get the prediction and the confidence interval with

```
> predict(threecars.lm1,newx,int="confidence")
```

4.5 Randomization Test for a Relationship

Example 4.9: Predicting GPAs with SAT Scores

Here is an R script for carrying out the randomization test (also called a permutation test) of the null hypothesis that the population correlation is zero.

```
SATGPA.df <- read.csv(file=file.choose()) #get the data!
attach(SATGPA.df)
cor(VerbalSAT,GPA) #should produce .2444543 as the sample r
x <- VerbalSAT
y <- GPA
originalr <- cor(x,y)
NEWy <- sample(GPA) #permutes the 24 GPA values
cor(x,NEWy) #get the permutation cor value
#We might want to repeat this a few times to see how the correlations vary
#now let's do this many times
N <- 1000 #set the number of simulation runs to 1000
permcorr <- as.numeric(0) #create a place to store results
for (i in 1:N){
NEWy <- sample(GPA)
permcorr[i] <- cor(x,NEWy)
}

permcorr[1:5] #look at the first 5 results
hist(permcorr) #make a histogram of the results
upper <- sum(permcorr>abs(originalr)) #count those results > 0.2444543
lower<- sum(permcorr<(-abs(originalr))) #count the lower tail results
#N.B.: "permcorr<-abs(originalr)" would _assign_ .2444543 to permcorr
# so we have #to be careful and use parentheses around abs(originalr)!
pvalue <- (upper+lower)/N
pvalue #see the simulation P-value
```

The line `hist(permcorr)` produces something similar to Figure 4.15 but without the shading of the two areas.

In the R code above we count the number of times that the permutation correlation is greater than the observed 0.244 and we make a separate count of the number of permutation correlations smaller than -0.244. This approach to conducting a two-sided test works well when the distribution of permutation results is reasonably symmetric, as can be seen in Figure 4.15.

An alternative approach that works even when the distribution is skewed is to count only the cases in the upper tail (when the observed test statistic is positive, as it is here) and then double the result to get the p-value for the nondirectional test. The R script below illustrates this.

```
upper <- sum(permcorr>originalr) #count those results > 0.2444543
pvalue <- upper*2/N
pvalue #see the simulation P-value
```

4.6 Bootstrap for Regression

Example 4.10: Porsche Prices

Here is an R script for collecting bootstrap samples.

```
PorschePrice.df <- read.csv(file.choose()) #get the data!
attach(PorschePrice.df)
originalmodel <- lm(Price~Mileage)
summary(originalmodel) #Look at the fitted model

car <- 1:30 #create indices 1,2,...,30
bootsample <- sample(car,replace=TRUE) #create a bootstrap sample
head(PorschePrice.df[bootsample,]) #look at the first few boot cases

bootmodel <- lm(Price~Mileage,data=PorschePrice.df[bootsample,])
#fit the model #using the bootstrap sample
summary(bootmodel) #look at the fitted model from the bootstrap sample

#Now do this many times
bootbetas <- matrix(0,nrow=5000, ncol=2) #set up a matrix to hold results

for (i in 1:5000) {   #the main bootstrap loop
     bootsample <- sample(car,replace=TRUE)
     bootmodel <- lm(Price~Mileage,
               data <- PorschePrice.df[bootsample,])
     bootbetas[i,] <- coef(bootmodel)
 }

hist(bootbetas[,2]) #Make a histogram of the slopes (the second coefficient)
```

To construct a confidence interval using Method #1 we need the SD of the bootstrap slopes, found with the command

```
sd(bootbetas[,2])
```

To construct a confidence interval using Method #2 we need quantiles of the bootstrap distribution. For this we use the commands

```
> quantile(bootbetas[,2],.025)
> quantile(bootbetas[,2],.975)
```

The quantiles of the bootstrap distribution of slopes are also used in Method #3.

CHAPTER 5

Analysis of Variance

5.1 The One-Way Model: Comparing Groups

We will follow the textbook's lead and use the fruit flies dataset to illustrate the use of R for the one-way ANOVA analysis.

Example 5.1: Fruit Flies

In the code below, we first obtain the dataset, define it to be the data frame **FruitFlies.df** and then confirm that we have 5 variables, with names given below, and we use functions `head` and `str` to see the structure of the data frame.

Looking at the output from `str` we see that the first 6 variables are either integer or numeric variables, but that the seventh variable—`Treatment`—is a *Factor* variable. A factor is R's variable type for categorical variables; they make it possible to give meaningful names for the values of a categorical variable. Also, some analyses will require R to distinguish whether a variable is numerical or categorical and thus whether the variable type is numerical or factor. This distinction is important for ANOVA.

```
> FruitFlies.df <- read.csv(file=file.choose())
> attach(FruitFlies.df)
> names(FruitFlies.df)

[1] "ID"        "Partners"  "Type"      "Longevity" "Thorax"    "Sleep"     "Treatment"

> head(FruitFlies.df)

  ID Partners Type Longevity Thorax Sleep  Treatment
1  1        8    0        35   0.64    22 8 pregnant
2  2        8    0        37   0.68     9 8 pregnant
3  3        8    0        49   0.68    49 8 pregnant
4  4        8    0        46   0.72     1 8 pregnant
```

```
5  5        8    0        63   0.72    23 8 pregnant
6  6        8    0        39   0.76    83 8 pregnant

> str(FruitFlies.df)

'data.frame':   125 obs. of  7 variables:
 $ ID       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Partners : int  8 8 8 8 8 8 8 8 8 8 ...
 $ Type     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Longevity: int  35 37 49 46 63 39 46 56 63 65 ...
 $ Thorax   : num  0.64 0.68 0.68 0.72 0.72 0.76 0.76 0.76 0.76 0.76 ...
 $ Sleep    : int  22 9 49 1 23 83 23 15 9 81 ...
 $ Treatment: Factor w/ 5 levels "1 pregnant","1 virgin",..: 3 3 3 3 3 3 ...
```

We now replicate some of the results given in the textbook, starting with basic descriptive analysis. The first code does the dotplots and the second code gives the summary statistics. Again, you will note that the textbook's Minitab dotplots have a different look but describe the same story.

```
> attach(FruitFlies.df)

> library(lattice)  # calls up the lattice library, which contains
                  # some graphing functions we need.

> xyplot(Longevity ~ Treatment) # produces comparative dotplots
```

In the code below, we replicate Table 5.1 from the text, using rounding in the second attempt to make the output easier to read.

```
# Assuming here FruitFlies.df is attached

> n <- tapply(Longevity,Treatment,length)
> mean <- tapply(Longevity,Treatment,mean)
> SD <- tapply(Longevity,Treatment,sd)
> cbind(n,mean,SD)  #combine the 3 vectors column-wise.


# NOTE:  We added the standard deviation to the summary statistics.
# Let's clean up the output by rounding and putting in textbook order.


> x <- c(5,1,3,2,4)
> n <- tapply(Longevity,Treatment,length)
> mean <- round(tapply(Longevity,Treatment,mean),2)
```

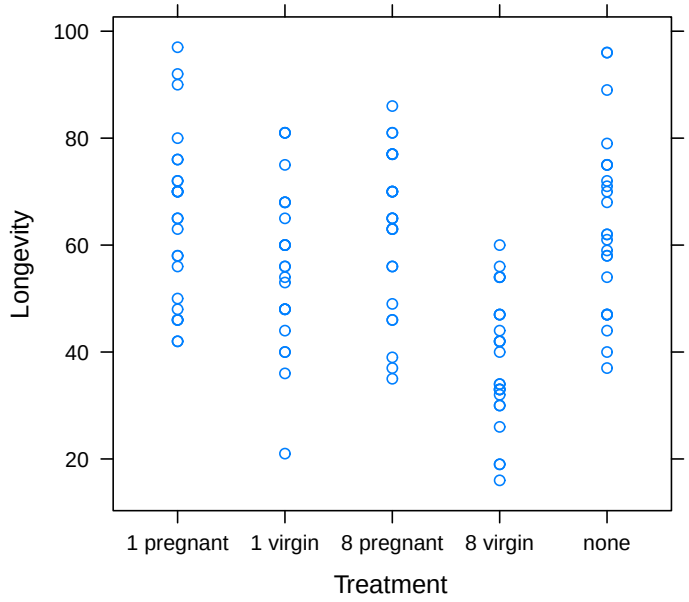


Figure 5.1 (page 223): Dotplot of life spans for fruit flies

```
> SD <- round(tapply(Longevity,Treatment,sd),2)
> summary.mat <- cbind(n,mean,SD)[x,]  # order rows using x
> summary.mat

            n  mean    SD
none       25 63.56 16.45
1 pregnant 25 64.80 15.65
8 pregnant 25 63.36 14.54
1 virgin   25 56.76 14.93
8 virgin   25 38.72 12.10
```

The textbook uses the `FruitFlies.df` data to illustrate fundamental concepts about the one-way ANOVA. We could use R's basic vector operators to construct the triple decomposition, a process that would use concepts you have learned about elsewhere in the *Companion*. Instead, since we want to concentrate in this section on the R methods new to ANOVA, we will get to the triple decomposition, residual plots, by first doing the inferential analysis of one-way ANOVA. This leads to Section 5.2 of the text.

5.2 Assessing and Using the Model for One-Way ANOVA

Here we introduce R's `aov` function for fitting an ANOVA model. The form of syntax and output for this function are similar to that of the `lm` function we learned about in our regression sections. The `aov` function models a numeric response variable on a set of predictors that are factors. We already encountered the use of factor predictors with regression—the case of dummy variables—so that the main difference with `aov` is that it expects only predictors that are factors.

The output of `aov` contains features that are special to the ANOVA setting but which are still amenable to the use of the `summary` function.

```
> attach(FruitFlies.df)
> FruitFlies.aov <- aov(Longevity ~ Treatment)
> summary(FruitFlies.aov)
             Df Sum Sq Mean Sq F value    Pr(>F)
Treatment     4  11939 2984.82  13.612 3.516e-09 ***
Residuals   120  26314  219.28
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

The following code reproduces Figures 5.3 (page 235) and 5.4 (page 236), residual plots for assessing the conditions of the one-way ANOVA model. Note the use of our `myqqnorm` function for Figure 5.3.

```
> myqqnorm(FruitFlies.aov$resid)  # Normal probability plot of residuals

> plot(FruitFlies.aov$fitted, FruitFlies.aov$resid) # Resids vs fits
> abline(h=0)                                       # add the x-axis
```

Transformations

We now proceed to use of the **Diamonds2** dataset to illustrate how to reproduce the results in the text on transformations in the ANOVA setting.

```
> Diamonds2.df <- read.csv(file=file.choose())

> str(Diamonds2.df)
'data.frame':   307 obs. of  6 variables:
 $ Carat     : num  1.08 0.31 0.32 0.33 0.33 0.35 0.35 0.37 0.38 0.38 ...
 $ Color     : Factor w/ 4 levels "D","E","F","G": 2 3 3 1 4 3 3 3 1 2 ...
 $ Clarity   : Factor w/ 8 levels "IF","SI1","SI2",..: 5 7 7 1 7 5 5 7 1 8 ...
 $ Depth     : num  68.6 61.9 60.8 60.8 61.5 62.5 62.3 61.4 60 61.5 ...
 $ PricePerCt: num  6693 3159 3159 4759 2896 ...
 $ TotalPrice: num  7229 979 1011 1570 956 ...
```

Assume we have ascertained, using diagnostic tools discussed previously in this chapter, that a log transformation is prudent. We accomplish this task with the following code.

```
> attach(Diamonds2.df)
> log.carat <- log(Carat)  # This is natural (base e) log
> Diamond.aov <- aov(log.carat ~ Color)
> summary(Diamond.aov)
             Df Sum Sq Mean Sq F value   Pr(>F)
Color         3  7.618 2.53918  12.742 7.28e-08 ***
Residuals   303 60.382 0.19928
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

5.4 Fisher's Least Significant Difference

We return now to the fruit flies data. We use the following code to calculate Fisher's LSD for the **FruitFlies** data.

```
> e <- FruitFlies.aov$resid
> MSE <- sum(e^2)/120
> t.crit <- qt(.975, 120)
> LSD <- t.crit*sqrt(MSE*(1/25 + 1/25))
> LSD
[1] 8.292658
```

Notice that this 8.29 value agrees with Example 5.10 of the textbook. To obtain all possible differences in mean values to decide which are statistically different from which, we use the code:

```
> diffs <- outer(mean,t(mean),"-")
> diffs
, , 1 pregnant

             [,1]
1 pregnant   0.00
1 virgin    -8.04
8 pregnant  -1.44
8 virgin   -26.08
none        -1.24
```

```
, , 1 virgin

             [,1]
1 pregnant   8.04
1 virgin     0.00
8 pregnant   6.60
8 virgin   -18.04
none         6.80

, , 8 pregnant

             [,1]
1 pregnant   1.44
1 virgin    -6.60
8 pregnant   0.00
8 virgin   -24.64
none         0.20

, , 8 virgin

            [,1]
1 pregnant 26.08
1 virgin   18.04
8 pregnant 24.64
8 virgin    0.00
none       24.84

, , none

             [,1]
1 pregnant   1.24
1 virgin    -6.80
8 pregnant  -0.20
8 virgin   -24.84
none         0.00
```

The `outer` function takes the matrices in the first two arguments and the function named in the third argument to produce an array. Here, the first argument is the vector `mean`, which is coerced into a matrix with 5 rows and 1 column. The second argument `t(mean)` takes the *transpose* of this matrix, which will have 1 row and 5 columns. Then, all possible differences are computed into an array of dimensions 5 by 1 by 5. Simply printing off `diffs` shows all possible differences with transparent labels, so that we can easily identify those that are significantly different.

CHAPTER 6

Multifactor ANOVA

6.1 The Two-Way Additive Model (Main Effects Model)

Since the goal of the *R Companion* is to introduce you to R, we will dispense here with the one-way analyses, since you learned those in the previous chapter. We go straight to the two-way model.

Example 6.1: Frantic fingers

Below we give R code to replicate the two-way analysis for this example.

```
Fingers.df <- read.csv(file=file.choose())  # Get the data.

> Fingers.df     # Print the data.

   Subject        Drug TapRate
1        I     Placebo      11
2       II     Placebo      56
3      III     Placebo      15
4       IV     Placebo       6
5        I    Caffeine      26
6       II    Caffeine      83
7      III    Caffeine      34
8       IV    Caffeine      13
9        I Theobromine      20
10      II Theobromine      71
11     III Theobromine      41
12      IV Theobromine      32
```

The structure of the dataset is the same, but the order stored in **Fingers** is different than given in Table 6.2 (page 276). Since Table 6.1 is a user-friendly way to view the data, we give R code to produce that here. We also produce the row and column means of the table, and attach them to the table.

```
> attach(Fingers.df)
> tab <- tapply(TapRate,list(Subject,Drug),mean) # create a table of means
> tab  # print out the table
    Caffeine Placebo Theobromine
I         26      11          20
II        83      56          71
III       34      15          41
IV        13       6          32


> row.means <- tapply(TapRate,Subject,mean) # compute row means

> col.means <- tapply(TapRate,Drug,mean) # compute column means

> rbind(cbind(tab,row.means),c(col.means,34)) # attach row and column means
                                              # to the table.
    Caffeine Placebo Theobromine row.means
I         26      11          20        19
II        83      56          71        70
III       34      15          41        30
IV        13       6          32        17
          39      22          41        34

> # Notice that there is no name for the row of column means.
> # We will rectify that.

> tab2 <- rbind(cbind(tab,row.means),c(col.means,34))

> row.names(tab2)
[1] "I"   "II"  "III" "IV"  ""
> row.names(tab2)[5] <- "col.means"
> tab2
          Caffeine Placebo Theobromine row.means
I               26      11          20        19
II              83      56          71        70
III             34      15          41        30
IV              13       6          32        17
col.means       39      22          41        34
```

If we would prefer to list Placebo first, as in the text, we use:

```
> tab2[,c(2,1,3)]
> tab2[,c(2,1,3)]
```

```
          Placebo Caffeine Theobromine
I              11       26          20
II             56       83          71
III            15       34          41
IV              6       13          32
col.means      22       39          41
```

In the code, the `rbind(cbind(tab,row.means),c(col.means,34))` creates a matrix with 5 rows and 4 columns, and then the `[,c(2,1,3)]` tells R to print out all rows—that's the empty spot before the comma—and then use the columns in the order 2, 1, and then 3, giving the order we want: Placebo, Caffeine, and then Theobromine.

The code below replicates the two-way additive model fit summarized in the text on page 280.

```
> attach(Fingers.df)
> Fingers.aov <- aov(TapRate ~ Drug + Subject)
> summary(Fingers.aov)
            Df Sum Sq Mean Sq F value    Pr(>F)
Drug         2    872  436.00  7.8795 0.0209669 *
Subject      3   5478 1826.00 33.0000 0.0003993 ***
Residuals    6    332   55.33
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

The following code reproduces the two plots in Figure 6.1.

```
# The following two give the normal quantile plot,
# with the same axis labels as the text.
> qqnorm(Fingers.aov$resid,ylab="Residuals",xlab="Normal Quantiles")
> qqline(Fingers.aov$resid)

# The following gives the residuals vs. fits plot:
> plot(Fingers.aov$fitted,Fingers.aov$resid,xlab="Fitted Tap Rate", ylab="Residuals")
> abline(h=0)
```

Inference after the Two-way ANOVA: The River Iron Example

We now replicate the river iron examples 6.2 and 6.3. We will first replicate the parts of these examples that we learned how to do above and then go on to obtain *interaction plots* and *Fisher's LSD*.

Example 6.2: River iron

```
> RiverIron.df <- read.csv(file = file.choose())
> RiverIron.df$FeLog10 <- log10(RiverIron.df$Fe) # define the logged variable
> attach(RiverIron.df)
> tab <- tapply(FeLog10,list(Site,River),mean)
> row.means <- tapply(FeLog10,Site,mean)
> col.means <- tapply(FeLog10,River,mean)

> tab2 <- rbind(cbind(tab,row.means),c(col.means,mean(FeLog10)))
> row.names(tab2)
[1] "DownStream" "MidStream"  "Upstream"   ""

> row.names(tab2)[4] <- "col.means" # Name the 4th row "col.means"

> round(tab2[c(3,2,1,4),],4)
           Grasse Oswegatchie Raquette St. Regis row.means
Upstream   2.9750      2.9345   2.0334    2.8756    2.7046
MidStream  2.7202      2.3598   1.5563    2.7543    2.3477
DownStream 2.5145      2.1139   1.4771    2.5441    2.1624
col.means  2.7366      2.4694   1.6889    2.7247    2.4049
```

Explanation of R code It might be useful to explicate the R code above. The `tapply` function creates a simple table of mean values of the `FeLog10` variable, given as the first argument. The second argument value, `list(Site,River)`, tells `tapply` to use the Site variable as the row variable and the River variable as the column variable. The third argument, `mean`, tells R to produce a mean value for each combination of Site and River. We assign this table to the `tab` variable, which is a 3-by-4 table of mean values.

We then use `tapply` again to create vectors of row and column means—assigning them to variables `row.means` and `col.means`.

The final lines of R code pack a lot of R power. We create the `tab2` table by adding the vector of `row.means` as a column using the `cbind` function; this operation is nested inside a call to the `rbind` function that adds a row to the bottom of the table and this row is a concatenation—using the `c` function—of the `col.means` vector to a single, final entry which is the grand mean of the logged iron values.

If any of this is confusing, we suggest implementing pieces of the R code starting from the inside and working out. For example, first try typing in `mean(FeLog10)`. Then type in `c(col.means,mean(FeLog10))`. And in this fashion, work your way to the outside, observing closely what answers R gives along the way.

To produce the interaction plot of Figure 6.3, we use the code below. We must fuss a bit with the order of the factor levels in order to get the plot to look like the textbook figure.

```
> # Now obtain the interaction plot.
> Site1 <- Site[12:1] # reverse the order of the Site factor
> levels(Site1) <- levels(Site)[3:1] # reverse the order of the level names
> interaction.plot(Site1, River, FeLog10)
```

We produce Figure 6.4 and the ANOVA table for the River Iron model with the R code below. We trust these lines of code should now seem familiar.

```
> RiverIron.aov <- aov(FeLog10 ~ River + Site)
> summary(RiverIron.aov)
            Df  Sum Sq Mean Sq F value    Pr(>F)
River        3 2.18703 0.72901  48.153 0.0001366 ***
Site         2 0.60765 0.30383  20.069 0.0021994 **
Residuals    6 0.09084 0.01514
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
> qqnorm(RiverIron.aov$resid)
> qqline(RiverIron.aov$resid)
> plot(RiverIron.aov$fitted, RiverIron.aov$resid)
> abline(h=0)
```

Finally, we can calculate the Fisher's LSD values for the two factors via the formulas

$$LSD_A = t^* \sqrt{\frac{2 \cdot MSE}{J}} \qquad \text{and} \qquad LSD_B = t^* \sqrt{\frac{2 \cdot MSE}{K}}$$

Here, Factor A is Site, so that $J = 4$ and Factor B is River, so that $K = 3$. When comparing sites with Fisher's LSD at a 5% level we use $LSD_{Site} = 2.45\sqrt{2 * 0.0151/4} = 0.213$ and when we compare rivers we use $LSD_{River} = 2.45\sqrt{2 * 0.0151/3} = 0.246$. The 2.45 value comes from the inverse CDF for the t-distribution with 6 degrees of freedom, that is, `qt(.975,6)`, the precise value being `2.446912`. The .0151 is the .01514 mean-square for error obtained from the ANOVA table. The rest of the story is told in the text.

6.2 Interaction in the Two-Way Model

We will illustrate the R calculations for the case of a balanced two-way ANOVA structure, with more than one observation per factor combination. We will take the **PigFeed** example. The book describes the new statistical issues that arise in this setting, and the R code uses nothing we have

not already seen before. The code below produces a table of means for each factor combination, an interaction plot (Figure 6.5), and the ANOVA output.

```
> PigFeed.df <- read.csv(file=file.choose())
> attach(PigFeed.df)
> tapply(WgtGain,list(Antibiotic, B12), mean)
    No Yes
No  19  22
Yes  3  54
> interaction.plot(B12,Antibiotic,WgtGain)
> PigFeed.aov <- aov(WgtGain ~ Antibiotic + B12)
> summary(PigFeed.aov)
            Df Sum Sq Mean Sq F value  Pr(>F)
Antibiotic   1    192  192.00  0.8563 0.37892
B12          1   2187 2187.00  9.7537 0.01226 *
Residuals    9   2018  224.22
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

CHAPTER 7

Additional ANOVA Topics

We now describe how to use R to implement the following additonal topics in ANOVA:

- Levene's test,
- three methods for multiple tests—Fisher's LSD, Bonferroni, and Tukey's HSD,
- comparisons and contrasts,
- ANOVA done through regression with indicators, and
- analysis of covariance.

7.1 Levene's Test for Homogeneity of Variances

Example 7.1: Checking equality of variances in the fruit fly data

Levene's test is available in the `car` library. The code below replicates Example 7.1:

```
> attach(FruitFlies.df)
> library(car)
> leveneTest(Longevity ~ Treatment)
Levene's Test for Homogeneity of Variance (center = median)
       Df F value Pr(>F)
group   4  0.4916 0.7419
```

The results agree with those given in the text, Minitab being the package that produced the output there.

7.2 Multiple Tests

We discussed Fisher's LSD in Chapter 5, but in this section we explain R code that will implement all three procedures, first using two built-in R functions, which have some limitations, and then

using our own code, which overcome these limitations.

The two built-in R functions that perform the three multiple comparisons methods under discussion are `pairwise.t.test` and `TukeyHSD`. The code below illustrates them, in turn.

The first code illustrates the `pairwise.t.test` function, which will compute p-values for Fisher's LSD and Bonfferoni's procedures, but not Tukey's LSD. The first argument is the response variable, the second is the explanatory, and the third is the method of adjustment for multiple comparisons. The results, obviously, agree with those obtained before, but this function provides no confidence interval option, which is a limitation.

```
Fisher's LSD p-values:

> pairwise.t.test(Longevity, Treatment, p.adj='none')
        Pairwise comparisons using t tests with pooled SD

data:  Longevity and Treatment

           1 pregnant 1 virgin 8 pregnant 8 virgin
1 virgin   0.057      -        -          -
8 pregnant 0.732      0.118    -          -
8 virgin   7.3e-09    3.4e-05  3.7e-08    -
none       0.768      0.107    0.962      3.0e-08

P value adjustment method: none

Bonfferoni p-values:
> pairwise.t.test(Longevity, Treatment, p.adj='bonf')
        Pairwise comparisons using t tests with pooled SD

data:  Longevity and Treatment

           1 pregnant 1 virgin 8 pregnant 8 virgin
1 virgin   0.57282    -        -          -
8 pregnant 1.00000    1.00000  -          -
8 virgin   7.3e-08    0.00034  3.7e-07    -
none       1.00000    1.00000  1.00000    3.0e-07

P value adjustment method: bonferroni
```

The next code uses the `TukeyHSD` function, which does give output for confidence intervals. Here we input the ANOVA object, `FruitFlies.aov`, rather than response and explanatory variables. Here the output is similar to the text's Minitab results, but in a less transparent configuration.

The limitation of this function is simply that it only gives us the Tukey results.

```
> TukeyHSD(FruitFlies.aov)
  Tukey multiple comparisons of means 95% family-wise confidence level

Fit: aov(formula = Longevity ~ Treatment)

$Treatment
                          diff        lwr        upr     p adj
1 virgin-1 pregnant     -8.04 -19.640468   3.560468 0.3126549
8 pregnant-1 pregnant   -1.44 -13.040468  10.160468 0.9969591
8 virgin-1 pregnant    -26.08 -37.680468 -14.479532 0.0000001
none-1 pregnant         -1.24 -12.840468  10.360468 0.9983034
8 pregnant-1 virgin      6.60  -5.000468  18.200468 0.5157692
8 virgin-1 virgin      -18.04 -29.640468  -6.439532 0.0003240
none-1 virgin            6.80  -4.800468  18.400468 0.4854206
8 virgin-8 pregnant    -24.64 -36.240468 -13.039532 0.0000004
none-8 pregnant          0.20 -11.400468  11.800468 0.9999988
none-8 virgin           24.84  13.239532  36.440468 0.0000003
```

The R code below is our own and provides a basic method for producing simultaneous confidence intervals with a family-wise error rate of α, and hence a family-wise coverage probability of $1 - \alpha$. We have varied below our tradition of including the $>$ R prompt, since this chunk of code was created from an R script and can be more easily typed into an R script (or text file) and executed as a single chunk, rather than line by line. Notice that the code produces confidence interval output that matches the text's Minitab results.

The term **script** in the previous paragraph has a technical meaning, which we describe here. When a sequence of R code becomes long, especially if it is likely to be used more than once, perhaps in modified form, it becomes cumbersome to enter the code via the command line. In this case, one can create a file of the code using a text editor and then this file can be entered into R directly at the `>` prompt to execute it in one fell swoop, using a copy-and-paste operation. While any text editor could be used for this purpose, implementations of R include a built-in editor to create these files and in this context R calls the file a script. R also gives you easy ways to run the script or portions of it. For example, you might look to the File menu and options like "New script" to create a new script or "Open script" to open an existing one. Then the menu bar will likely have a way to easily run the script (or portions of it) under one of the menu bar headings, such as Edit.

```
attach(FruitFlies.df)
names(FruitFlies.df)
n <- length(unique(Treatment)) # n=number of Treatments
k <- choose(n,2) # k=number of pairs
```

```
# Produce Fisher LSD intervals:
     # means.tab = vector of Treatment means
means.tab <- as.vector(tapply(Longevity,Treatment,mean))
     # n.tab = vector of sample sizes
n.tab <- as.vector(tapply(Longevity,Treatment,length))
     # summary.df will contain the CI information
     # we initialize it to all 0s.
summary.df <- as.data.frame(matrix(0,ncol=8,nrow=k))
names(summary.df) <- c("i","j","n.i","n.j",         # attach names to columns
                  "i-j.diff","lcl","ucl","signif?")
     # create a simple set of Treatment names
name <- c("1pg","1vir","8pg","8vir","none")
mse <- 219.28
dfe <- 120
alpha <- .05
     # We compute mean differences and confidence limits
     # through a pair of nested "for loops."
index <- 0
for (i in 1:(n-1)) {
   for (j in (i+1):n)  {
     index <- index + 1
     summary.df[index,1] <- name[i]
     summary.df[index,2] <- name[j]
     summary.df[index,3] <- n.tab[i]
     summary.df[index,4] <- n.tab[j]
     center <- means.tab[i] - means.tab[j]
     summary.df[index,5] <- center
     moe <- qt(1-alpha/2, dfe)*sqrt(mse*(1/n.tab[i] + 1/n.tab[j]))
     # The next two lines are commented out, but would replace the
     # previous line for producing Bonferroni or Tukey HSD intervals.
     # moe <- qt(1-alpha/(2*k), dfe)*sqrt(mse*(1/n.tab[i] + 1/n.tab[j]))
     # moe <- (qtukey(1-alpha,n,dfe)/sqrt(2))*sqrt(mse*(1/n.tab[i] + 1/n.tab[j]))
     lcl <- center - moe
     ucl <- center + moe
     summary.df[index,6] <- lcl
     summary.df[index,7] <- ucl
     if (lcl*ucl > 0) summary.df[index,8] <- "YES" else summary.df[index,8] <- "no"
     }}

> summaryLSD.df <- summary.df
```

```
Fisher's LSD Output:
> summaryLSD.df
      i    j n.i n.j i-j.diff         lcl        ucl signif?
1   1pg 1vir  25  25     8.04  -0.2526709  16.332671      no
2   1pg  8pg  25  25     1.44  -6.8526709   9.732671      no
3   1pg 8vir  25  25    26.08  17.7873291  34.372671     YES
4   1pg none  25  25     1.24  -7.0526709   9.532671      no
5  1vir  8pg  25  25    -6.60 -14.8926709   1.692671      no
6  1vir 8vir  25  25    18.04   9.7473291  26.332671     YES
7  1vir none  25  25    -6.80 -15.0926709   1.492671      no
8   8pg 8vir  25  25    24.64  16.3473291  32.932671     YES
9   8pg none  25  25    -0.20  -8.4926709   8.092671      no
10 8vir none  25  25   -24.84 -33.1326709 -16.547329     YES
```

Explanation of code We create a data frame called `summary.df` to store the results of our calculations; we begin by initializing it to all zeroes. Constants `n` and `k` are the number of treatments (5) and the number of pairs of treatments (10). We use the `comb` function to calculate `k` as "combinations of `n` things taken 2 at a time" or `k <- choose(n,2)`.

Next, we enter a set of 8 column names for `summary.df` and 5 names for the different treatments, using names shorter than given in the book, but easy to identify with those in the book. We shorten them to make the output easier on the eye. We next enter the mean square error, its degrees of freedom, and the value of α.

We now enter the looping stage of the code. The `for` command in R creates a sequence of lines that are repeated for various instances of the "loop counter," which is `i` for the first `for` statement and `j` for the next one. The first encountered loop—called the outer loop—runs through values of `i` beginning at 1 and ending at `n-1=5-1=4`.

For each value of `i`, the second loop—the inner loop—runs through values of `j` beginning at `i+1` and ending at `n=5`. The result of these "nested for loops" is that we repeat the sequence of R lines within the loops as we run through the `(i,j)` combinations: (1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5).

Each time through this sequence we compute the entries to that particular `(i,j)` combination of treatments with the first column being the i-th treatment name, the second column being the j-th treatment name, the third and fourth columns being the two sample sizes, the fifth column being the difference in sample means ("i minus j"), the sixth column being the lower confidence limit (`lcl`) and the seventh column being the upper confidence limit (`ucl`). Finally, the eighth column records a YES for those pairs that are statistically significantly different, and otherwise it records a "no."

In the code we denote the margin of error of the confidence interval by `moe`. The only difference between our three methods lies in the calculation of `moe`. In the code we have commented out two lines that give alternative calculations for the margins of error for Bonferroni and Tukey methods. To run these alternatives, simply comment out the current `moe` line for Fisher's LSD and uncomment the alternative you want to calculate.

Having done separate calculations for all three methods we now add the output given below to give us all three procedures. Note the agreement with the textbook's Minitab results. (Some subtractions of group means were made in reverse order, creating negative signs for positive signs, and vice versa.)

```
Bonfferoni Output:
> summaryBONF.df
      i    j n.i n.j i-j.diff        lcl        ucl signif?
1   1pg 1vir  25  25     8.04  -3.938157  20.018157      no
2   1pg  8pg  25  25     1.44 -10.538157  13.418157      no
3   1pg 8vir  25  25    26.08  14.101843  38.058157     YES
4   1pg none  25  25     1.24 -10.738157  13.218157      no
5  1vir  8pg  25  25    -6.60 -18.578157   5.378157      no
6  1vir 8vir  25  25    18.04   6.061843  30.018157     YES
7  1vir none  25  25    -6.80 -18.778157   5.178157      no
8   8pg 8vir  25  25    24.64  12.661843  36.618157     YES
9   8pg none  25  25    -0.20 -12.178157  11.778157      no
10 8vir none  25  25   -24.84 -36.818157 -12.861843     YES

Tukey HSD Output:
> summaryTukey.df
      i    j n.i n.j i-j.diff        lcl        ucl signif?
1   1pg 1vir  25  25     8.04  -3.560486  19.640486      no
2   1pg  8pg  25  25     1.44 -10.160486  13.040486      no
3   1pg 8vir  25  25    26.08  14.479514  37.680486     YES
4   1pg none  25  25     1.24 -10.360486  12.840486      no
5  1vir  8pg  25  25    -6.60 -18.200486   5.000486      no
6  1vir 8vir  25  25    18.04   6.439514  29.640486     YES
7  1vir none  25  25    -6.80 -18.400486   4.800486      no
8   8pg 8vir  25  25    24.64  13.039514  36.240486     YES
9   8pg none  25  25    -0.20 -11.800486  11.400486      no
10 8vir none  25  25   -24.84 -36.440486 -13.239514     YES
```

7.3 Comparisons and Contrasts

The code below reproduces two contrasts discussed in the textbook for the fruit flies data, Examples 7.5 and 7.7. In essence, we are simply using R as a convenient calculator. Particular points to note in the calculation are:

- the use of `levels(Treatment)` to remind ourselves of the order of the treatments, so we can define the contrast vector correctly;
- the use of `tapply` to obtain vectors of means and sample sizes for the 5 treatments;
- the use of the elements `residuals` and `df.residual` from the `FruitFlies.aov` object, which we use to calculate the mean square error term (`mse`); and
- the use of vector arithmetic in computing the value of the sample contrast (`effect1`) and the standard error of the sample contrast (`se1`).

  We include the calculation for the second contrast as well, although the details work exactly the same way.

```
> attach(FruitFlies.df)
> levels(Treatment)   # We want to recall the Treatment names
                      # and their order.
# [1] "1 pregnant" "1 virgin"   "8 pregnant" "8 virgin"   "none"
> means <- as.vector(tapply(Longevity,Treatment,mean))
# Vector of treatment means
> n <- as.vector(tapply(Longevity, Treatment, length))
# Vector of treatment sample sizes
> mse <- sum(FruitFlies.aov$residuals^2/FruitFlies.aov$df.residual) # compute MSE
> con1 <- c(0,0,0,1,-1) # This is the contrast we wish to calculate.
> effect1 <- sum(con1*means) # This computes the sample contrast, a simple linear
                           # combination of treatment means using the contrast.
> se1 <- sqrt(sum(mse*con1^2/n)) # Compute the standard error for the contrast.
> t1 <- effect1/se1  # Compute the t-ratio for the contrast.
> 2*pt(t1,FruitFlies.aov$df.residual)  # Computer the one si

# [1] 2.979631e-08 <--- the p-value is nearly 0

> con2 <- sum(c(-.5,.5,-.5,.5,0)*means)
> effect2 <- sum(con2*means)
> se2 <- sqrt(sum(mse*con2^2/n))
> t2 <- effect2/se2
> 2*pt(t2,FruitFlies.aov$df.residual)

# [1] 3.575887e-75 <--- the p-value is nearly 0
```

7.4 Nonparametric Statistics

Two-Sample Nonparametric Procedures

Here we introduce the R implementation of the Wilcoxon-Mann-Whitney procedure which depends upon the `wilcox-test` function. We begin by asking for R help about this function using `help(wilcox.test)`. We reproduce a portion of the help message. First, `wilcox.test(x, ...)` indicates that only the first argument, a vector denoted by `x` here, is required. But the next portion of the help message tells us that for a two-sample problem we will need a second vector, `y`. Since the text has produced a confidence interval we need to define the `conf.int = TRUE` argument to override the default, which is to not produce a confidence interval. Also note that the default procedure is to correct for ties, as indicated by the `correct=TRUE` argument.

The R code below reproduces the results from the textbook Example 7.11. First, we obtain the dataset, attach it, and verify the names of the variables; we are interested in the second two, which give tail length for red-tailed (RT) and sharp-shinned hawks (SS). We then use `summary` to verify the basic summary statistics and note that while median values match up, the sample sizes are not right. Notice the 316 missing values for the sharp-shinned hawks. This has resulted from having extracted the two vectors from a data frame and data frames always are row-and-column structures, with equal column lengths. The length of 577 for the red-taileds made 577 the column dimension, so the short-fall in sharp-shinned tail-lengths resulted in 316 missing values (NAs) being entered in that column.

We remedy this imbalance by defining a new vector `Tail_SS_nonNA` that just contains the non-missing values. Notice that we now have sample sizes and medians matching those in the text. From there, we use `wilcox.test` to compute the Wilcoxon-Mann-Whitney test, having now two numeric vectors with no missing values. And we see results that agree with those in the book.

```
# Asking for help is a sign of maturity:

> help(wilcox.test)

# What follows is a portion of the help message:
Usage

wilcox.test(x, ...)

## Default S3 method:
wilcox.test(x, y = NULL,
            alternative = c("two.sided", "less", "greater"),
            mu = 0, paired = FALSE, exact = NULL, correct = TRUE,
            conf.int = FALSE, conf.level = 0.95, ...)
```

```
> HawkTail2.df <- read.csv(file=file.choose())
> attach(HawkTail2.df)
> names(HawkTail2.df)
[1] "Tail_CH" "Tail_RT" "Tail_SS"
> length(Tail_RT)
[1] 577
> length(Tail_SS)
[1] 577

> summary(Tail_RT)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  122.0   214.0   221.0   222.1   230.0   288.0
> summary(Tail_SS)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  119.0   133.0   150.0   146.7   158.0   221.0   316.0

# Note the missing values for the Sharp-shinneds. We remove those:

Tail_SS_nonNA <- Tail_SS[is.na(Tail_SS)==0]

> Tail_SS_nonNA <- Tail_SS[is.na(Tail_SS)==0]
> summary(Tail_SS_nonNA)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  119.0   133.0   150.0   146.7   158.0   221.0

> wilcox.test(Tail_RT,Tail_SS_nonNA,conf.int=T,alternative ="two.sided")

> wilcox.test(Tail_RT,Tail_SS_nonNA,conf.int=T)

        Wilcoxon rank sum test with continuity correction

data:  Tail_RT and Tail_SS_nonNA
W = 149305, p-value < 2.2e-16
alternative hypothesis: true location shift is not equal to 0
95 percent confidence interval:
 74.00000 78.00004
sample estimates:
difference in location
              76.00005
```

Nonparametric ANOVA

The R `kruskal.test` function calculates the Kruskal-Wallis procedure that the book introduces, from Example 7.12. We again include an R help message followed by code that reproduces the book results. There are two ways to input the data to `kruskal.test`: (1) as a list of data vectors, or (2) as a formula of the form `y ~ x`, where the `y` variable gives the data values and the `x` variable gives the group values. We will use both forms to reproduce the *Cancer Survival* results.

After obtaining and attaching the Cancer Survival data, we compute the Kruskal-Wallis test to obtain the same values that the text reports. We also produce a table of sample size and median by group, to conform to the results in the book. We have included the boxplot R command to reproduce Figure 7.6 but have not actually included the figure in this document.

We next, redo the example using the first of the options for input listed above. This example actually lends itself more naturally to the second option above, so we first define some separate group vectors for each of the 5 types of cancer. Then we enter these as a list of 5 vectors to be compared with the Kruskal-Wallis test. The results are identical, of course.

```
kruskal.test(x, ...)

## Default S3 method:
kruskal.test(x, g, ...)

## S3 method for class 'formula':
kruskal.test(formula, data, subset, na.action, ...)

Arguments
x  a numeric vector of data values, or a list of numeric data vectors.

g  a vector or factor object giving the group for the corresponding elements of x.
Ignored if x is a list.

formula a formula of the form lhs ~ rhs where lhs gives the data values and rhs the
corresponding groups.

data  an optional matrix or data frame (or similar: see model.frame) containing
the variables in the formula formula. By default the variables are taken from
environment(formula).
subset  an optional vector specifying a subset of observations to be used.
na.action  a function which indicates what should happen when the data contain
NAs. Defaults to getOption("na.action").

...  further arguments to be passed to or from methods.
```

```
------------------------------------------------------------
> # Cancer Survival Example

> CancerSurvival.df <- read.csv(file=file.choose()) # obtain Cancer Survival data
> names(CancerSurvival.df)
> attach(CancerSurvival.df)
> boxplot(Survival ~ Organ) # re-produce boxplot
> kruskal.test(Survival ~ Organ)

        Kruskal-Wallis rank sum test

data:  Survival by Organ
Kruskal-Wallis chi-squared = 14.9539, df = 4, p-value = 0.004798

# Obtain sample sizes and medians by group:

> med <- tapply(Survival,Organ,median)
> n <- tapply(Survival,Organ,length)
> cbind(n,med)
          n  med
Breast   11 1166
Bronchus 17  155
Colon    17  372
Ovary     6  406
Stomach  13  124

> # Alternative input to kruskal.test

# First, define separate survival vectors for each of the 5 groups:
> Breast <- Survival[Organ=="Breast"]
> Bronchus <- Survival[Organ=="Bronchus"]
> Colon <- Survival[Organ=="Colon"]
> Ovary <- Survival[Organ=="Ovary"]
> Stomach <- Survival[Organ=="Stomach"]

# Now, apply kruskal.test to the list comprising these 5 vectors:
> kruskal.test(list(Breast,Bronchus,Colon,Ovary,Stomach))

        Kruskal-Wallis rank sum test

data:  list(Breast, Bronchus, Colon, Ovary, Stomach)
Kruskal-Wallis chi-squared = 14.9539, df = 4, p-value = 0.004798
```

7.5 ANOVA and Regression with Indicators

We learned about regression with indicators as predictors in Section 4.4. We will illustrate here how to code up the example from the current section that uses indicators to analyze some simple ANOVA designs that, on first encounter, do not appear to be regression problems.

Two-Sample Comparison of Means as Regression

The code below recalculates the pooled two-sample t-test for Example 7.13. Let's go through the various parts of the code. After attaching `FruitFlies.df` we form the subset of this data frame corresponding to just the two treatments—"8 virgin" and "none"—required for the example. The `indices` variable we create in the second line contains 125 true or false values, where TRUE indicates a row number we wish to be in our data subset. Note the use of the logical operator | for the logical "or." The result of the logical "or" is to put a TRUE value into `indices` if either treatment value is true. (You can print out `indices` to verify this.) Then, in line 4, we create the two-groups data frame by taking only those rows of `FruitFlies.df` for which indices have a TRUE value. Again, the blank space after the comma in `FruitFlies.df[indices,]` indicates that all columns are chosen.

After detaching `FruitFlies.df` we attach `FruitFliesTwoGroups.df` to simplify references to the variable names. The line that defines the variable `Eight` takes all values of the `Longevity` variable that correspond to a treatment value of "8 virgin." Similarly we create the vector of Longevity values called `none`. The two vectors `Eight` and `none` are the two variables we place into the `t.test` function to reproduce the two-sample t-test, using the `var.equal=T` argument to obtain the pooled t-test. Notice results that agree with those in the text.

```
> attach(FruitFlies.df)
> indices <- FruitFlies.df$Treatment=="8 virgin" | FruitFlies.df$Treatment=="none"
> detach()
> FruitFliesTwoGroups.df <- FruitFlies.df[indices,]
> attach(FruitFliesTwoGroups.df)
> Eight <- Longevity[Treatment=="8 virgin"]
> none <- Longevity[Treatment=="none"]
> t.test(Eight,none,var.equal=T)

        Two Sample t-test

data:  Eight and none
t = -6.0811, df = 48, p-value = 1.885e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -33.05298 -16.62702
```

```
sample estimates:
mean of x mean of y
    38.72     63.56
```

Next, we redo the calculations as a simple regression problem with a single indicator as the predictor variable. The logical statement `Treatment == "8 virgin"` creates a TRUE-FALSE vector, whereas we would like our dummy predictor variable to be a numeric 0-1 variable. We could coerce the logical into a numeric with a `as.numeric(Treatment=="as.virgin")` function call, but the `lm` function does this automatically for us. Again, we see agreement with the text.

```
> detach()
> attach(FruitFliesTwoGroups.df)
> v8 <- Treatment == "8 virgin"
> model <- lm(Longevity ~ v8)
> summary(model)

Call:
lm(formula = Longevity ~ v8)

Residuals:
   Min     1Q Median     3Q    Max
-26.56  -8.72  -1.56   8.40  32.44

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   63.560      2.888  22.006  < 2e-16 ***
v8           -24.840      4.085  -6.081 1.88e-07 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 14.44 on 48 degrees of freedom
Multiple R-squared: 0.4352,     Adjusted R-squared: 0.4234
F-statistic: 36.98 on 1 and 48 DF,  p-value: 1.885e-07
```

One-Way ANOVA for Means as Regression

Here, we use all 5 treaments for the fruit flies and compare the ANOVA approach to the regression. (See Example 7.14.) The ANOVA calculations with R follow and are straightforward. We note the agreement with the results in the textbook.

```
attach(FruitFlies.df)
model <-aov(Longevity ~ Treatment)
summary(model)
```

```
> summary(model)
             Df Sum Sq Mean Sq F value    Pr(>F)
Treatment     4  11939 2984.82  13.612 3.516e-09 ***
Residuals   120  26314  219.28
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

We next replicate with R the equivalent regression calculations. After attaching the data frame (which would not be needed if this code follows that above), we use `levels(Treatment)` to cause R to print the values of the `Treatment` factor. We proceed to define the 4 indicator or dummy variables that we will use in our regression model. These are the variables `p1`, `v1`, `p8`, `v8`. For example, as we observed above, the statement `p1 <- Treatment=="1 pregnant"` defines a logical vector (TRUEs and FALSEs) of length 125 saying if each case has a Treament value of "1 pregnant" or not. After defining these 4 indicators, we compute the model with the `lm` function and obtain a summary. Again, the results agree with those in the textbook.

```
> attach(FruitFlies.df)
> levels(Treatment)
[1] "1 pregnant" "1 virgin"   "8 pregnant" "8 virgin"   "none"

> p1 <- Treatment=="1 pregnant" # creates indicator variable for "1 pregnant"
> v1 <- Treatment=="1 virgin" # creates indicator variable for "1 virgin"
> p8 <- Treatment=="8 pregnant" # creates indicator variable for "8 pregnant"
> v8 <- Treatment=="8 virgin" # creates indicator variable for "8 virgin"
> model <- lm(Longevity ~ p1 + v1 + p8 + v8)
> summary(model)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   63.560      2.962  21.461  < 2e-16 ***
p1TRUE         1.240      4.188   0.296    0.768
v1TRUE        -6.800      4.188  -1.624    0.107
p8TRUE        -0.200      4.188  -0.048    0.962
v8TRUE       -24.840      4.188  -5.931 2.98e-08 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 14.81 on 120 degrees of freedom
Multiple R-squared: 0.3121,     Adjusted R-squared: 0.2892
F-statistic: 13.61 on 4 and 120 DF,  p-value: 3.516e-09
```

Two-Way ANOVA for Means as Regression

The code below replicates the two-way ANOVA for the PigFeeds data, Example 7.15.

```
> attach(PigFeed.df)
> names(PigFeed.df)
> model.aov <- aov(WgtGain~Antibiotic+B12)
> summary(model.aov)


            Df Sum Sq Mean Sq F value  Pr(>F)
Antibiotics  1    192  192.00  0.8563 0.37892
B12          1   2187 2187.00  9.7537 0.01226 *
Residuals    9   2018  224.22
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

To perform the equivalent regression analysis, and reproduce the text's results, we use the R code below. The first two lines define the indicator predictors: A has value 1 when a case has a value of "Yes" for Antibiotics, 0 otherwise; B has value 1 when the value of B12 is "Yes," 0 otherwise. We use the `as.numeric function` to coerce a True-False vector into a 1-0 vector, although if we did not use `as.numeric` R would automatically coerce the TRUEs to 1s and the FALSEs to 0s, so in that sense using `as.numeric` is optional.

We use `summary(model.lm)` to obtain the regression output that agrees with that in the textbook, and we use `anova(model.lm)` to obtain the ANOVA table with sums of squares, the difference being that the textbook's sum of squares for regression is the sum of R's separate sum of squares for the A and B variables.

```
#Two predictor regression with no interaction:

> A <- as.numeric(Antibiotics=="Yes") # creates indicator variable for Antibiotics; Yes=1
> B <- as.numeric(B12=="Yes")  # creates indicator variable for B12; Yes=1

> model.lm <- lm(WgtGain ~ A + B)
> summary(model.lm)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    7.000      7.487   0.935   0.3742
A              8.000      8.645   0.925   0.3789
B             27.000      8.645   3.123   0.0123 *
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
Residual standard error: 14.97 on 9 degrees of freedom
Multiple R-squared: 0.5411,     Adjusted R-squared: 0.4391
F-statistic: 5.305 on 2 and 9 DF,  p-value: 0.03006

> anova(model.lm)
Analysis of Variance Table

Response: WgtGain
          Df Sum Sq Mean Sq F value  Pr(>F)
A          1    192  192.00  0.8563 0.37892
B          1   2187 2187.00  9.7537 0.01226 *
Residuals  9   2018  224.22
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

We now proceed to fit the model with interaction, first replicating the two-way ANOVA results using the `aov` function and following this with the equivalent regression model using the `lm` function.

```
> model2.aov <- aov(WgtGain ~ Antibiotics + B12 + Antibiotic:B12)
> summary(model2.aov)

               Df Sum Sq Mean Sq F value    Pr(>F)
Antibiotics     1    192  192.00  5.2966  0.050359 .
B12             1   2187 2187.00 60.3310 5.397e-05 ***
Antibiotics:B12 1   1728 1728.00 47.6690  0.000124 ***
Residuals       8    290   36.25
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

We then reproduce the corresponding analysis as a regression model with an interaction term that is the product of indicators A and B. The results agree again with the textbook.

```
> model2.lm <- lm(WgtGain ~ A + B + A*B)
> summary(model2.lm)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   19.000      3.476   5.466 0.000597 ***
A            -16.000      4.916  -3.255 0.011619 *
B              3.000      4.916   0.610 0.558624
A:B           48.000      6.952   6.904 0.000124 ***
---
```

```
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 6.021 on 8 degrees of freedom
Multiple R-squared: 0.934,      Adjusted R-squared: 0.9093
F-statistic: 37.77 on 3 and 8 DF,  p-value: 4.532e-05


> anova(model2.lm)
Analysis of Variance Table

Response: WgtGain
          Df Sum Sq Mean Sq F value    Pr(>F)
A          1    192  192.00  5.2966  0.050359 .
B          1   2187 2187.00 60.3310 5.397e-05 ***
A:B        1   1728 1728.00 47.6690  0.000124 ***
Residuals  8    290   36.25
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Analysis of Covariance

We begin by reproducing the Grocery Stores example, Example 7.17. First, the one-way ANOVA:

```
> Grocery.df <- read.csv(file=file.choose())
> names(Grocery.df)
> model1.aov <- aov(Sales ~ Discount)
> summary(model1.aov)
            Df Sum Sq Mean Sq F value Pr(>F)
Discount     2   1288  644.19  0.5734 0.5691
Residuals   33  37074 1123.46

> library(car)
> leveneTest(Sales ~ Discount)
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  2  0.9262 0.4061
      33
#Using our earlier user-defined function myqqnorm:
> myqqnorm(model1.aov$resid)

# Alternatively, using basic R functions:
> qqnorm(model1.aov$resid)
> qqline(model1.aov$resid)
```

```
# Either produce the results from the text.

# Finally the code to produce a residuals-vs-fits plot:
> plot(model1.aov$fit, model1.aov$resid)
```

As in the text, note the nonsignificant relationship between `Sales` and `Discount`. The plot in Figure 7.11(a) gives a good visual confirmation of this result. We give here the R code for that figure. A couple of nuances deserve explanation. The first line uses the `levels` argument to the `factor` function to define a more reasonable order to the factor levels than the default order, which is alphabetical. Yes, "10.00" comes before "5.00" as alphabetical strings, but alpha-order is not natural here and is not the order we want for our comparative dotplots.

```
# Without the first line, the plot gives the groups in the order 10.00%, 15.00%, 5.00%,
# which is alphabetical, but unnatural in this context.

> Grocery.df$Discount <- factor(Grocery.df$Discount,levels=c("5.0%","10.00%","15.00%"))
> attach(Grocery.df)
> library(lattice)
> xyplot(Sales ~ Discount,col="black")  # The default color is blue.
```

Figure 7.10 tells us that the conditions of normally distributed errors (Figure (a)) and equal variances (Figure (b)) are met when considering just the one-way ANOVA model of `Sales` on the factor of interest, `Discount`. This establishes the first of the three conditions for the ANCOVA model.

We next go to Figure 7.11, which gives a scatterplot of `Sales` versus `Price`, controlling for `Discount`. The R code is given here. The explanation is as follows. After attaching the dataset, we use `plot` to graph the full scatterplot of all 36 data points, except that by using the `type='n'` argument, we suppress the actual plotting of the points. The reason for this line is that we wish to set up our axes, with proper labels, and with x and y ranges sufficient for all points in the dataset. We suppress the plotting of points because we want separate scatterplots for the subsets obtained by controlling for discount level.

The next section of code defines 3 new x-variables and 3 corresponding y-variables based upon the value of `Discount`. For example, the line `Sales5 <- Sales[Discount=="5.00%"]` defines a y-variable that is all those values of `Sales` for which the value of `Discount` is 5.00%. The next section of code uses the `points` function to add points to the existing coordinate system that creates the 3 scatterplots, each at a different discount value, each using a different plotting symbol. We then use the `locator` function to interactively add a legend in the upper left-hand corner of the plot, this location chosen by the user because it is a large area of open white space.

The final section of code adds regression lines to each of the 3 scatterplots using the `lm` function to compute the line and the `abline` function to graph the line. Each line gets a different line style using the `lty` argument.

```
> attach(Grocery.df)
> plot(Price, Sales,xlab="Price",ylab="Sales",type="n") # set up axes

# This group of scatterplots defines 3 subsets of the data based
# upon discount level:
> Sales5 <- Sales[Discount=="5.00%"]
> Price5 <- Price[Discount=="5.00%"]
> Sales10 <- Sales[Discount=="10.00%"]
> Price10 <- Price[Discount=="10.00%"]
> Sales15 <- Sales[Discount=="15.00%"]
> Price15 <- Price[Discount=="15.00%"]

# This section of code produces 3 separate scatterplots for the 3 subsets.
> points(Price5,Sales5,pch=1) # open circle
> points(Price10,Sales10,pch=0) # open square
> points(Price15,Sales15,pch=16) # solid circle

# This allows the user to place a proper legend, interactively, into the plot:
> legend(locator(1),c("5.00%","10.00%","15.00%"), pch = c(1,0,16))

# These 3 lines of code add regression lines to each of the 3 scatterplots:
> abline(lm(Sales5 ~ Price5), pch=1)
> abline(lm(Sales10 ~ Price10), lty=2)
> abline(lm(Sales15 ~ Price15), lty=4)
```

As stated in the text, this figure lends support to the appropriateness of the ANCOVA model, in that there is a clear shift in level depending upon the level of the discount, but within each discount level the relationship between `Sales` and `Price` is roughly linear and the three slopes tend to be about the same.

Before proceeding with the ANCOVA calculations, we reproduce the plots (Figure 7.12 and Figure 7.13) that corroborate the second ANCOVA condition that the regression conditions for a simple regression of `Sales` on `Price` should be met, ignoring `Discount`. This is straightforward code similar to some we saw in the earliest regression examples.

```
> attach(Grocery.df)
> plot(Price, Sales)  # A basic scatterplot
> model <- lm(Sales ~ Price) # Compute the simple linear regression line
> qqnorm(model$resid)  # normal plot of residuals
> qqline(model$resid)  # add a line to the normal plot

> plot(model$fit, model$resid)  # create a residuals-vs-fit plot
> abline(h=0,lty=2) # for clarity, add dashed horizontal line at 0
```

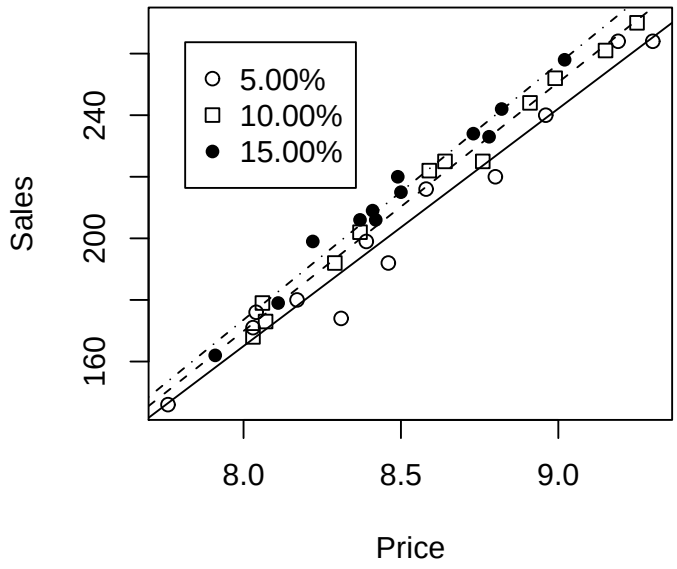


Figure 7.11 (page 371): Scatterplot of sales against price by amount of discount

Finally, we recompute Example 7.19, "Grocery using ANCOVA model" with R. Here is the code. *Note:* To calculate the ANCOVA analysis the order of predictors is important in the model statement, so we use the order `Price + Discount`, rather than the other way around. This is because the `anova` function computes sequential sums of squares and we want the effect of `Discount` after adjusting for `Price`. The results below do confirm those in the text.

```
> model <- lm(Sales ~ Price + Discount)
> anova(model)

Analysis of Variance Table
Response: Sales
          Df Sum Sq Mean Sq  F value    Pr(>F)
Price      1  36718   36718 1391.366 < 2.2e-16 ***
Discount   2    800     400   15.149 2.348e-05 ***
Residuals 32    844      26
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Notice then that the ANOVA table, created by `anova(model)`, agrees with that in the text, except that R does not give us the values of `S` or R^2. To obtain these requires the statement `summary(model)`. We also point out that the `summary(model)` output gives coefficient estimates for the indicator variables that R automatically creates when it recognizes that `Discount` is a factor with three levels. (The choice of the 5.00% discount level as the reference category was also determined by R.) Here is the R output.

```
> summary(model)

Call:
lm(formula = Sales ~ Price + Discount)

Residuals:
    Min      1Q  Median      3Q     Max
-14.444  -3.183   1.050   3.510   9.045

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)    -472.953     18.317 -25.820  < 2e-16 ***
Price            79.591      2.148  37.052  < 2e-16 ***
Discount10.00%    6.822      2.107   3.238  0.00280 **
Discount15.00%   11.476      2.098   5.471 5.04e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 5.137 on 32 degrees of freedom
Multiple R-squared: 0.978,      Adjusted R-squared: 0.9759
F-statistic: 473.9 on 3 and 32 DF,  p-value: < 2.2e-16
```

CHAPTER 9

Logistic Regression

9.1 Choosing a Logistic Regression Model

Example 9.1: Losing Sleep The data on teen sleep was collected using a survey that coded 0=No and 1=Yes as shown in Table 9.2.

To summarize this raw data in a table like Table 9.1,
use the `table` command.

```
table(sleep, age)
# if you want to refer to the table later, give it a name
sleep.tab = table(sleep, age)
```

Alternatively, if you have data summarized in a table, you can create the table in R. That's the approach we take here, but things are somewhat simplified if you work with the raw data. We create the summary table, Table 9.1, for the teens' sleep data from Example 9.1.

```
> sleep.mat=matrix(c(12,34,35,79,37, 77,39,65,27,41),nrow=2,
                     dimnames=list(c("Less than 7 hours", "7 hours or more"),
                                           c("14","15","16","17","18")))
>
> sleep.mat
                   14 15 16 17 18
Less than 7 hours 12 35 37 39 27
7 hours or more   34 79 77 65 41
```

If you would like to include the column totals you can use the `addmargins` command.

```
> addmargins(sleep.mat,1)
                   14  15  16  17 18
Less than 7 hours 12  35  37  39 27
7 hours or more   34  79  77  65 41
Sum               46 114 114 104 68
```

Graphs with Proportion of Yes's

The next three graphs concerning the sleep data have the proportion of teens getting 7 or more hours of sleep. It is easy to create these proportions with the data summarized in our table using `prop.table`. Note that a table must be input to the `prop.table` command. Here, we have named our table; sometimes we create the table within the `prop.table` command, for example, `prop.table(table(sleep,age),2)`.

```
# Table of proportion of no's and yes's for each age
prop.tab=round(prop.table(sleep.mat,2),3) #age-conditional probability of yes
prop.tab
```

Figure 9.1 Proportion of Yes's by Age

These proportions can be plotted versus age, we only need to plot the proportion of yes's for each age group. Figure 9.1 can be produced by the code below.

```
> age=14:18
> prop.yes=prop.tab[2,]  # the second row of the table of proportions
> plot(age,prop.yes,xlab="Age",ylab="Proportion Saying Yes",
                             ylim=c(0.5,.9),xlim=c(12,20))
```

Figure 9.2 Proportion of Yes's with a Regression Line by Age

It's tempting to create a regression line to add to the plot of age versus the proportion of yes's along with horizontal lines at 0 and 1. We do that here but come up with a much better approach later. The dotted lines represent the bounds for a proportion, 0 and 1.

```
plot(age,prop.yes,xlab="Age",ylab="Proportion Saying Yes",
                            ylim=c(-0.1,1.1),xlim=c(0,40))
 abline(lm(prop.yes~age))
 abline(h=1,lty=2)
 abline(h=-0,lty=2)
```

Figure 9.3 Proportion of Yes's with Logistic Regression Curve by Age

In place of using a regression line, a line based on the predicted values from a logistic regression model is used to relate the proportion saying yes to age. To obtain the curve in Figure 9.3, first we use logistic regression to produce predicted responses. Using the option `type="response"` produces predicted probabilities.

```
l.model=glm(cbind(sleep.mat[2,],sleep.mat[1,])~age, family=binomial)
newage.df=data.frame(age=0:40)
# Generating predicted values using the original logistic reg based on 5 proportions
yhat=predict(l.model, newdata=newage.df,type="response")
plot(newage.df$age,yhat, pch='',type="l",ylim=c(-0.1,1.1),xlim=c(0,40),
       xlab="age", ylab="Proportion Saying No")
points(age,prop.yes)
 abline(h=1,lty=2)
 abline(h=-0,lty=2)
```

Odds and Log(odds)

Plotting the logits versus age requires the logits be calculated. The odds are used to calculate the logits after which it is very straightforward to produce the logits with R.

```
# Logit transformed probabilities vs Age
odds = (prop.yes)/(1-prop.yes)
round(odds,4)   # look at the odds
logit = log(odds)
round(logit,4)           # look at the logits

> round(logit,4)     # look at the logits
    14     15     16     17     18
1.0408 0.8142 0.7309 0.5108 0.4180
```

Figure 9.4 Logit versus Age

Figure 9.4 can now be constructed as we have done in many plots previously, we have calculated the logits by age.

```
plot(age,logit, ylim=c(-3,3), xlim=c(0,40),
                           ylab="logit(Proportion Saying Yes)", xlab="Age")
abline(lm(logit~age))
```

Table 9.3 Various values of π, odds, and log(odds)

Table 9.3 includes several values of π, the corresponding odds $= \pi/(1-\pi)$, and the log(odds) $= \log(\pi/(1-\pi))$. Table 9.3 illustrates the calculation of odds and log(odds) for several different proportions π. It can be reproduced in part using the following R code.

```
> # before the command that assigns pi to a vector of values from 1/20 to 19/20,
> # when we type pi
> pi
[1] 3.141593

pi=c(1/20,1/10,1/5,1/4,1/2,3/4,4/5,9/10,19/20)
> pi  # what you get is what you expected
[1] 0.05 0.10 0.20 0.25 0.50 0.75 0.80 0.90 0.95

odds=pi/(1-pi)

> round(odds,3)   # look at the odds
[1]  0.053  0.111  0.250  0.333  1.000  3.000  4.000
[8]  9.000 19.000

logit = log(odds)
round(logit, 3)           # look at the logits
[1] -2.944 -2.197 -1.386 -1.099  0.000  1.099  1.386  2.197  2.944
```

Figure 9.6 The Logistic Transformation

The logistic transformation is depicted in Figure 9.6.

```
plot(pi,logit,pch='',type="l",xlim=c(0,1),ylim=c(-4,4),
                                    xlab="Probability", ylab="log odds")
```

Table 9.4 Values of π, odds, and log(odds)

We have already produced the second and third rows in Table 9.4 using R, naming them `prop.yes` and `logit`. The fourth and fifth rows in this table use our fitted logistic regression based on the observations for ages 14 through 18 to obtain estimated or fitted values, $\hat{\pi}$, for ages 2 through 30. R can help us out with this.

```
l.model=glm(cbind(sleep.mat[2,],sleep.mat[1,])~age, family=binomial)
agenew.df=data.frame(age=c(2,12,14:18,20,30))
# Generating predicted values using the original logistic reg based on 5 proportions

xb.fit=predict(l.model, newdata=agenew.df)
xb.fit    # this is row four, the estimated logits, i.e. the linear predictor bo+b1x
y.fit=predict(l.model, newdata=agenew.df, type="response")
y.fit    # this is row five, the estimated probabilities
         # These are obtained by specifying type="response"
```

Figure 9.7 Observed log odds $\hat{p}_i$ versus *age* along with the fitted line

To construct Figure 9.7 using R, plot the estimated logits by age.

```
plot(agenew.df$age, xb.fit, xlab="Age",
ylab="logitProportion saying Yes",ylim=c(-3,3), xlim=c(0,40))
points(age, logit,pch=19)  #pch=19 creates solid circles
abline(lm(xb.fit~agenew.df$age))
```

Example 9.4 Medical School: Two Versions of the Logistic Regression Model

We next take up with Example 9.4, the medical school admission example. We enter and attach the data frame MedGPA.df. We produce Figure 9.8, the jittered scatterplot, using the following R code. The nuance lies in the use of the `jitter` function. To jitter a scatterplot means to add some random noise to the coordinates of the points, the goal being to allow us to separate points that have the same values and would therefore be impossible for us to see as multiple points. Since the y-variable of `Acceptance` is binary, there is much overlap in the points unless we jitter them a bit. (There are instances where we might also want to jitter in the x-direction, but this is not such a case since `GPA` has few replicates.) The `amount` argument controls the amount of variation in the jitter. We chose `amount=.05` which means that R will randomly allow deviations with a range of $\pm.05$ of the true values (of 0 and 1). This seemed to give a result similar to the textbook's.

```
> MedGPA.df <- read.csv(file=file.choose())  # choose MedGPA.csv
> attach(MedGPA.df)
> names(MedGPA.df)

[1] "Accept" "Acceptance" "Sex"  "BCPM"  "GPA"  "VR"  "PS"  "WS"
[9] "BS"  "MCAT"  "Apps"

> table(Acceptance)
Acceptance
 0  1
25 30
```

The key new R concept in the area of logistic regression is the use of the `glm` function. The `glm` name is an abbreviation for "generalized linear model," a very flexible and general statistical method for fitting models to response-explanatory variable situations that do not conform to the somewhat strict conditions of the linear model or `lm` function. So, if error terms are non-normal or if variance is not constant, one will often find the `glm` function to be useful.

We use the `glm` function with the `family=binomial` argument to obtain the logistic regression model for predicting the probability of acceptance to medical school (the response variable) from the student's GPA (the explanatory variable). *Note:* For the response variable we use the numeric 0-1 variable `Acceptance` rather than the character variable `Accept`. The `glm` function demands a numeric response variable.

Note the `summary(Med.glm1)` command produces the regression style output that agrees with the Minitab output in the text. To produce the confidence interval for the odds ratio we add the command `confint(Med.glm1)`. Notice that the confidence interval for the odds ratio that R produces differs from that produced by Minitab. This is because R uses a more sophisticated procedure to calculate a confidence interval for the slope.

Figure 9.8(a) Data for ordinary and logistic regression

The two graphs in Figure 9.8:

```
> plot(GPA, MCAT)   # no need for jitter here, values of GPA differ
> abline(lm(MCAT~GPA))  # add the regression line

> med.glm1=glm(Acceptance~GPA, family=binomial)
> b0=coef(med.glm1)[1]
> b1=coef(med.glm1)[2]
> summary(med.glm1)

> plot(GPA, jitter(Acceptance,amount=.05),ylab="Acceptance")
        # the amount argument controls the
        # level of vertical variation.
> curve(exp(b0+b1*x)/(1+exp(b0+b1*x)),add=T)



# Make sure you use Acceptance rather than Accept for the response variable.
> Med.glm1 <- glm(Acceptance ~ GPA, family=binomial)
> summary(Med.glm1)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7805  -0.8522   0.4407   0.7819   2.0967

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -19.207      5.629  -3.412 0.000644 ***
GPA            5.454      1.579   3.454 0.000553 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 56.839  on 53  degrees of freedom
AIC: 60.839
```

Later in this section of the text, confidence limits are computed for the coefficient associated with GPA. Minitab output included in the text uses the estimate of the coefficient, 5.45417 ± 1.96*1.57931 which is exponentiated to give a confidence interval for the odds ratio. Alternatively, R users can use the `confint` command to obtain a confidence interval for the coefficient on GPA. These values are close but differ slightly from the Minitab limits. However, for the confidence interval for the odds ratio, a different algorithm is used and it yields limits on the odds ratio that appear to be quite different from Minitab. The Minitab interval given in the textbook is 10.58 to 5164.74, while that given below by R is 14.83 to 7829.25. The Minitab interval uses the Wald statistic given in the summary table and a simple standard normal approximation, while R uses a fancier method chosen by R's statistical computing experts. It is hard to say which result to put one's faith in and this difference is a fairly large shift to the right between the Minitab and the R. (Note: The second line below about "profiling" is some clue to the method R uses.)

```
> confint(Med.glm1)
Waiting for profiling to be done...
                 2.5 %    97.5 %
(Intercept) -31.713266 -9.376103
GPA           2.696316  8.965621

> exp(confint(med.glm1))
                    2.5 %       97.5 %
(Intercept) 1.686955e-14 8.472476e-05
GPA         1.482501e+01 7.829246e+03
```

9.2 Logistic Regression and Odds Ratios

Odds and Odds Ratios

In Example 9.6 on zapping migraines we already discussed odds ratios in logistic regression, using both the `glm` and the `confint` functions. If one has simple two-by-two tables such as in Examples 9.6, 9.7, or 9.8, then it is almost simpler to just calculate odds ratios with a hand-held calculator. We can always use R as a calculator, as given below.

First, we create the table called `zap.tab`. In R, a table is a data type, just as data frames, vectors, or objects created by a call to something like the `lm` function are. In particular, a table has features that distinguish it from a matrix, even though it resembles a matrix and shares some characteristics such as dimension. In the first line of code, we create a matrix of the 4 numbers using the `matrix` function, but use `as.table` to interpret it as a table instead. Any table includes row and column names or headings, which are character strings. The default names here would be "A" and "B" for each, but we have more useful names in mind. The second line of code defines the row headings to be "Pain Free" and "Not Pain Free"; the third line of code creates the column headings of "TMS" and "Placebo." The next line of code simply asks R to do the arithmetic to compute the odds ratio. Note, for example, that `zap.tab[1,1]` simply refers to the (1,1) entry of the table, which is

the 39. Finally, the last line of code defines a function `or` (standing for odds ratio) that assumes a simple two-by-two table or matrix, called `x` as the argument and computes the odds ratio of it. We include one function call to illustrate we get the same results as we get by hand.

```
> zap.tab <- as.table(matrix(c(39,61,22,78),nrow=2))
> dimnames(zap.tab)[[1]] <- c("Pain Free","Not Pain Free")
> dimnames(zap.tab)[[2]] <- c("TMS","Placebo")
> zap.tab
              TMS Placebo
Pain Free      39      22
Not Pain Free  61      78

> is.table(zap.tab)
[1] TRUE

> zap.tab[1,1]*zap.tab[2,2]/(zap.tab[1,2]*zap.tab[2,1])

[1] 2.266766



# Define an odds ratio function; x is 2-by-2 table or matrix.
> or <- function(x) x[1,1]*x[2,2]/(x[1,2]*x[2,1])

> or(zap.tab) # Use new function on previous example.
[1] 2.266766  # Note agreement.
```

Slope and Odds Ratios when the Predictor is Quantitative

Next we take up Example 9.10, the golf-putting example. After entering the data frame with the `read.csv` function, we form the tabular summary in Table 9.5 using the `table` function. The `table` function cross-tabulates the data using the variables given in the two arguments. The first variable listed becomes the row variable, and the second becomes the column variable.

```
> Putts1.df <- read.csv(file=file.choose())
> names(Putts1.df)
[1] "Length" "Made"

> attach(Putts1.df)
> table(Made,Length)  #Create a table by cross-tabulating Made and Length
    Length
Made  3  4  5  6  7
   0 17 31 47 64 90
   1 84 88 61 61 44
```

If we insist on making the table look more like Table 9.5 we can use the code below. In defining the object `a` we create the same table as above, but reversing the two rows using the `c(2,1)` selection. We then create a vector of column sums using the `apply` function; the `2` value in the second argument tells R to perform a column-wise operation, and the `sum` in the third argument tells R to use addition as the operation; hence, column sums. Then we create the table named `tab` using `rbind`. This function row-binds the two inputs, meaning it combines the table `a` with the vector `b` by appending `b` as a third row, exactly as we want. The final step changes the row names of `tab` to more transparent names.

```
> a <- table(Made,Length)[c(2,1),] #a is the table with rows reversed
> b <- apply(a,2,sum) # find the column sums of the table
> tab <- rbind(a,b) # add the column sums as the bottom row of the table
> row.names(tab) <- c("successes","failures","total") # re-define row names
> tab # print out the table

            3   4   5   6   7
successes  84  88  61  61  44
failures   17  31  47  64  90
total     101 119 108 125 134
```

Next, we compute the logistic model and assign it the name `Putts.glm1` using the `glm` function. The results of `summary(Putts.glm1)` agree with those in the text, which were computed with R. We also include the confidence interval for the odds ratio, obtained with `confint(Putts.glm1)`. We don't include the Minitab output here or in the text, but this time the confidence intervals are, to two decimal places, identical between the two software packages.

```
> Putts.glm1 <- glm(Made ~ Length, family=binomial)
> summary(Putts.glm1)

Call:
glm(formula = Made ~ Length, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8705  -1.1186   0.6181   1.0026   1.4882

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.25684    0.36893   8.828   <2e-16 ***
Length      -0.56614    0.06747  -8.391   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 800.21  on 586  degrees of freedom
Residual deviance: 719.89  on 585  degrees of freedom
AIC: 723.89

Number of Fisher Scoring iterations: 4

> exp(confint(Putts.glm1))  # confint produces CIs for beta, so
                            # we need to exponentiate that interval.

                2.5 %     97.5 %
(Intercept) 12.7974573 54.4505172
Length       0.4960611  0.6464444
```

Medical School Example: GPA*10

As noted in the text, a metric for GPA which will produce more reasonable values for the odds ratio and its confidence limits can be found by multiplying GPA by 10. Now we find that although the confidence limits for the odds ratios differ between Minitab and R, the difference is not quite so dramatic. The Minitab limits in the text are 1.27 to 2.35, while the R limits below are 1.31 to 2.45. The R interval is only slightly shifted to the right this time.

```
> med.glm2=glm(Acceptance~GPA10, family=binomial)
> summary(med.glm2)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7805  -0.8522   0.4407   0.7819   2.0967

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -19.2065     5.6287  -3.412 0.000644 ***
GPA10         0.5454     0.1579   3.454 0.000553 ***
---
    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 56.839  on 53  degrees of freedom
AIC: 60.839
```

```
> confint(med.glm2)

                  2.5 %     97.5 %
(Intercept) -31.7132662 -9.3761026
GPA10         0.2696316  0.8965621
> exp(confint(med.glm2))

                   2.5 %       97.5 %
(Intercept) 1.686955e-14 8.472476e-05
GPA10       1.309482e+00 2.451162e+00
```

9.3 Assessing the Logistic Regression Model

Empirical Logit Plots

We now consider Figure 9.13. Figure 9.13 is disappointing as a diagnostic tool, as with just 55 data points 9 bins cannot adequately convey the characteristic logistic shape we would like. This disappointment evinces the book's assertion that "diagnostic plots will not be nearly as useful in logistic regression as we found them to be in ordinary regression." There can be occasions when such a plot could work better, for example, when the dataset is large. We present here code to produce our figure. (Crawley[page 597] provides code for an empirical logit plot that is defined slightly differently from ours.)

The R code below reproduces Table 9.6 and the right-hand figure in Figure 9.13. Before explaining the code, it might be useful to understand better how the two R functions `sort` and `order` work.

Consider the small example below. `foo` is a vector of integers as shown. The `sort` function creates a new vector in increasing order using the elements from `foo`. The `order` function gives the positions or indices for a vector of length 6 where one can find the sorted version. For example, the first element in `order(foo)` is 3, which indicates that the smallest element of `foo` occurs in position 3. The next element in `order(foo)` is 2, which indicates that the second smallest element of `foo` occurs in position 2. The next element in `order(foo)` is 6, which indicates that the third smallest element in `foo` is in position 6. And so on. Now we can see why the code `foo[order(foo)]` returns the sorted vector: the `order(foo)` argument gives the subscripts of `foo` that we want extracted. And `order` creates the vector of subscripts 1 through 6 in precisely the order that sorts the vector.

```
> foo
[1] 6 2 1 5 4 3
> sort(foo)
[1] 1 2 3 4 5 6
> order(foo)
[1] 3 2 6 5 4 1
> foo[order(foo)]
[1] 1 2 3 4 5 6
```

Now, we can better understand that first, important step in the following code. We create a version of the `MedGPA.df` data frame that is sorted, row-wise, by the `GPA` variable. The rest of the code creates a matrix of values, `x.mat`, of which the first row is the smallest 11 values of GPA, the second row is the next smallest, and so on. Then we compute means, number of yes, number of no, proportions, adjusted proportions, and logits to essentially replicate the table. From there we make the scatterplot and the added line.

```
# First, form a version of MedGPA.df, sorted by the GPA variable.
# We use the order function
sorted.MedGPA.df <- MedGPA.df[order(MedGPA.df$GPA),]
x <- sorted.MedGPA.df$GPA
y <- sorted.MedGPA.df$Acceptance
x.mat <- matrix(x,ncol=11,nrow=5, byrow=T)
x.means <- apply(x.mat,1,mean)
y.mat <- matrix(y,ncol=11,nrow=5, byrow=T)
y.yes <- apply(y.mat,1,sum)
y.no <- 11-y.yes
y.prop <- y.yes/(y.yes + y.no)
y.prop.adj <- (.5+y.yes)/(1+y.yes+y.no)

y.logit.adj <- log(y.prop.adj/ (1-y.prop.adj))
plot(x.means,y.logit.adj,xlab="GPA", ylab = "adjusted logit")
abline(lm(y.logit.adj ~ x.means))
```

9.4 Formal Inference: Tests and Intervals

We proceed to describe the R code for producing the formal inference for the 2 examples of this section. From the output we produce with this simple R code, the various by-hand calculations given in the text are at your fingertips.

First, we look at Example 9.4, using `GPA10` as the predictor, where two lines produce the basic logistic model output.

```
> attach(MedGPA.df)
> model1 <- glm(Acceptance ~ GPA10, family=binomial)
> summary(model1)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -19.2065     5.6287  -3.412 0.000644 ***
GPA10         0.5454     0.1579   3.454 0.000553 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 56.839  on 53  degrees of freedom
AIC: 60.839
```

Next, we do a similar thing for Example 9.10, the golf example.

```
> attach(Putts1.df)
> model2 <- glm(Made ~ Length, family=binomial)
> summary(model2)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.25684    0.36893   8.828   <2e-16 ***
Length      -0.56614    0.06747  -8.391   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 800.21  on 586  degrees of freedom
Residual deviance: 719.89  on 585  degrees of freedom
AIC: 723.89
```

Binary Predictors

Finally, we end with the zapping migraines example. We do not have the raw data, but only the data in the form of a 2-by-2 table, which we have called `zap.tab`. This is sufficient to build the appropriate R code, and we give it, along with output, below. Notice the results agree with the Minitab results from the text.

```
> zap.tab
              TMS Placebo
Pain Free      39      22
Not Pain Free  61      78

> Treatment <- c(1,0)
> model <- glm(cbind(zap.tab[1,],zap.tab[2,]) ~ Treatment, family=binomial)

> summary(model)
```

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -1.2657     0.2414  -5.243 1.58e-07 ***
Treatment     0.8184     0.3167   2.584  0.00977 **
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 6.8854  on 1  degrees of freedom
```

CHAPTER 10

Multiple Logistic Regression

10.1 Overview

We now describe the use of R for multiple logistic regression topics. In a way, the methods we learn here will reflect the kind of transition we saw in going from simple linear regression to multiple linear regression. The main function for fitting models will still, however, be `glm`.

Our focus in this chapter will be on fitting models from which one can assess models with the nested LRT test and in this way build to a good final model. (The exception to this will be an early digression to show how one can control graphical input in R.) The text shows a series of empirical logit plots for the purpose of building conceptual understanding. Since our exercises do not require those of you, we will dispense with them in this chapter. (We gave an example in the previous chapter, along with a reference to Crawley, should you want to explore this more on your own.)

10.2 Choosing, Fitting, and Interpreting Models

We begin with Example 10.1 about the *Corporate Average Fuel Economy* (CAFE) bill. The code below defines the data frame `CAFE.df`, attaches it, and prints out variable names. We then, in that fourth line of code, replicate Figure 10.1, the comparative dotplot. We have suppressed including this plot in this document, because we want to use this opportunity to digress to an illustration of how R can fine tune a graph. (You might want to look at the simple graph before proceeding.) The main problems with the simple graph are:

- The points overlap so much it is hard to distinguish them;
- We prefer black to blue plotting symbols; and
- Like the text, we prefer to exclude the lone Independent Senator from the graph.

```
> CAFE.df <- read.csv(file=file.choose())  # choose CAFE.csv
> attach(CAFE.df)
> names(CAFE.df)
[1] "Senator" "State" "Party" "Contribution" "LogContr" "Dem" "Vote"

> library(lattice)
> xyplot(Contribution ~ Party,col=1)  # First, unsatisfactory attempt: graph not included!
```

The `col=1` argument will change the plotting symbol to black. To eliminate the sole Independent Senator, we define new x and y vectors (of length 99, rather than 100) that expunge the offending Senator; note the use of the `!` symbol in the extraction statement `Party[!Party=="I"]`, standing for the logical *not* operation. That is, `x` is defined as all elements in the `Party` vector that are not Independents, and `y` is defined as all elements of `Contribution` that are not associated with that Independent Senator. We then ask for a plot of just Democrats and Republicans, but we decided to suppress this graph, so we can add some jitter in the horizontal direction to allow us to visually separate points better.

```
> x <- Party[!Party == "I"]
> length(Party)
[1] 100
> length(x)
[1] 99
> y <- Contribution[!Party == "I"]
> length(y)
[1] 99
> xyplot(y ~ x,col=1)
```

There is a problem with adding jitter in the x-direction: the x-variable is not numeric. Thus, we decide to switch from the `xyplot` function to the `plot` function so we can employ techniques we have seen before. First, we plot a jittered version of the `x` variable but interpret it as a numeric vector, using the `as.numeric` function. The result of `as.numeric(x)` would be to change Democrats to 1s, Independents (of which there are none in `x`) to 2s, and Republicans to 3s. If we did a simple plot of jittered x versus y we would get noninformative numeric x-axis labels, which we do not want.

By using the `xlab` and `ylab` arguments, we get axis labels to our liking. The `xaxt="n"` argument suppresses the x-axis labelling of tick marks, which would have been those noninformative numeric values 1, 2, and 3. Then, we follow up with a call to the `axis` function that attaches the label `Dem` where the number 1 would be—think of the 1 as there, even though its printing was suppressed—an empty label at the number 2, and a `Rep` label at the number 3. Finally, the `tck=0` argument just tells R that we prefer not to have tick marks; technically, we have drawn tick marks of length 0. All of this jittering and axis control is, perhaps, a bit fussy, but we include it to illustrate, once again, the power of R to exert fine control over graphs. The reader can also view this fussiness as

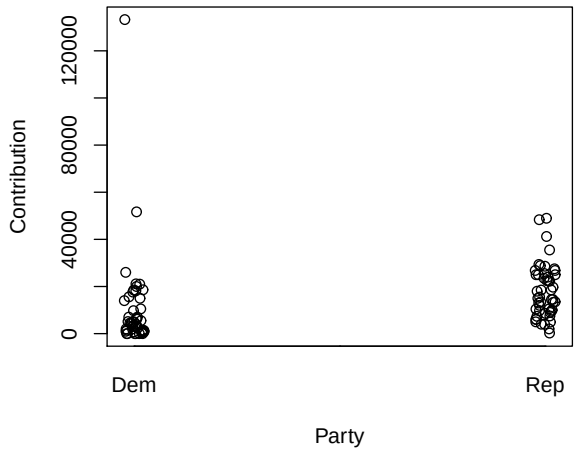


Figure 10.1: *Contributions by Party*

optional and opt instead for the initial plot. Notice, again, that the dotplot we obtain looks quite different from a Minitab dotplot.

```
> plot(jitter(as.numeric(x),amount=.05),y,col=1,
        xlab='Party',ylab='Contribution',xaxt="n")
> axis(1,1:3, labels=c("Dem","","Rep"),tck=0)
```

We now do some model fitting.

```
> model1 <- glm(Vote ~ log(1+Contribution) + Dem, family=binomial)
> summary(model1)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.2370  -0.8185   0.4291   0.5514   2.9891

Coefficients:
                     Estimate Std. Error z value Pr(>|z|)
(Intercept)           -2.5547     1.8651  -1.370 0.170769
log(1 + Contribution)  0.4833     0.1964   2.460 0.013884 *
Dem                   -1.9012     0.5594  -3.399 0.000677 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 132.813  on 99  degrees of freedom
Residual deviance:  93.226  on 97  degrees of freedom
AIC: 99.226
```

Is there an interaction?
We next replicate the model with an interaction term. We include the code below. The `I()` in the model definition turns out to be unnecessary here because the model contains the lower order linear terms that would be implied by the special meaning of the `*` if the expression had just been as `LogContr*Dem`. We include the `I()` because we want to remind ourselves that here we are literally thinking of variable multiplication.

```
> CAFE.glm2 <- glm(Vote ~ LogContr + Dem + I(LogContr*Dem), family=binomial)
> summary(CAFE.glm2)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.5711  -0.6419   0.3023   0.5631   2.2532

Coefficients:
                  Estimate Std. Error z value Pr(>|z|)
(Intercept)        -10.164      5.401  -1.882   0.0599 .
LogContr             3.002      1.357   2.212   0.0270 *
Dem                  2.544      5.974   0.426   0.6703
I(LogContr * Dem)   -1.088      1.515  -0.719   0.4724
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 132.813  on 99  degrees of freedom
Residual deviance:  86.781  on 96  degrees of freedom
AIC: 94.781

Number of Fisher Scoring iterations: 5
```

10.4 Formal Inference: Tests and Intervals

We use `model1` from the CAFE example above to compute tests and confidence intervals. We repeat the output from the fitted model and then use `confint` to compute confidence intervals. The salient command below is `exp(confint(model1))`. Notice that the confidence intervals differ slightly from those given in the book. The book intervals are using the Wald z-statistic while

`confint` uses a method called profile likelihood. The latter method is generally more trustworthy in cases where the two answers differ. So, for example, the 95% confidence intervals for the `Dem` coefficient are -2.997 to -0.805 in the text and -3.0639695 to -0.8428995 below. These are not wildly different but still maybe different enough to question which one to prefer.

```
Coefficients:
                      Estimate Std. Error z value Pr(>|z|)
(Intercept)            -2.5547     1.8651  -1.370 0.170769
log(1 + Contribution)   0.4833     0.1964   2.460 0.013884 *
Dem                    -1.9012     0.5594  -3.399 0.000677 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 132.813  on 99  degrees of freedom
Residual deviance:  93.226  on 97  degrees of freedom

# confidence intervals using "profile likelihood" rather than Wald z-statistic
> confint(model1)
Waiting for profiling to be done...
                           2.5 %     97.5 %
(Intercept)           -6.5903459  0.6331929
log(1 + Contribution)  0.1572700  0.9136299
Dem                   -3.0639695 -0.8428995
```

We turn now to Example 10.9 and give the code for the rather complicated model entertained there. Here we give the code to create that output.

```
> model <- glm(Acceptance ~ Fem + GPA + I(GPA^2)
                           + Fem:GPA + Fem:I(GPA^2),family=binomial)
> summary(model)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8864  -0.8632   0.1388   0.6758   2.2370
```

```
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)   53.4790    55.2279   0.968    0.333
Fem            9.7948   162.0299   0.060    0.952
GPA          -37.0418    32.8472  -1.128    0.259
I(GPA^2)       6.1276     4.8571   1.262    0.207
Fem:GPA       -5.9204    93.4748  -0.063    0.949
Fem:I(GPA^2)   0.9993    13.4642   0.074    0.941

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 52.066  on 49  degrees of freedom
```

We now proceed to Example 10.12, where several models are fit for the medical school acceptance data in order to build a good, final model. The fits in this example suffice to cover any model-building situations you will typically encounter. The text explains how to use the output to perform the desired nested LRT tests. Since the *R Companion's* goal is to teach about R, we will show a short series of model fits, with corresponding output, just to illustrate how one can get the consituent pieces (which the book illustrates quite fully) for doing nested LRT tests and model building.

(a) One predictor.

The code below provides the one of the three models fitting to a single predictor. We chose the GPA predictor.

```
> model1 <- glm(Acceptance ~ GPA, family=binomial)
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -19.207      5.629  -3.412 0.000644 ***
GPA            5.454      1.579   3.454 0.000553 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 56.839  on 53  degrees of freedom
AIC: 60.839
```

(b) Adding a second predictor.

Next, we create a couple of two-predictor models. The `model2` below uses `GPA` and `MCAT` in an additive two-predictor model. The residual deviance of 54.014 is the full model deviance in the

book, and the reduced model deviance of 56.839 is the residual deviance from the previous `model1` above.

```
> model2 <- glm(Acceptance ~ GPA + MCAT, family=binomial)
> summary(model2)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.7132  -0.8132   0.3136   0.7663   1.9933

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -22.3727     6.4538  -3.467 0.000527 ***
GPA           4.6765     1.6416   2.849 0.004389 **
MCAT          0.1645     0.1032   1.595 0.110786
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 54.014  on 52  degrees of freedom
```

In `model3` we again encounter the `I( )` operator to indicate to R that we want the mathematical function of squaring a variable.

```
> model3 <- glm(Acceptance ~ GPA + I(GPA^2), family=binomial)
> summary(model3)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8363  -0.8020   0.3207   0.7830   1.9553

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   33.332     45.275   0.736    0.462
GPA          -24.752     26.366  -0.939    0.348
I(GPA^2)       4.325      3.832   1.128    0.259

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 55.800  on 52  degrees of freedom
AIC: 61.8
```

We then fit a model that is additive in three predictors and call this `model4`.

```
> model4 <- glm(Acceptance ~ GPA + MCAT + Fem, family=binomial)
> summary(model4)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.0326  -0.8439   0.2524   0.6130   2.1607

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -25.2431     7.2019  -3.505 0.000456 ***
GPA           5.1392     1.8508   2.777 0.005491 **
MCAT          0.1809     0.1080   1.675 0.093946 .
Fem           1.2580     0.7303   1.723 0.084965 .
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 50.786  on 51  degrees of freedom
AIC: 58.786
```

Finally, we fit `model5`, using the notation `Fem:MCAT` for the interaction term. (Alternatively, one can use `I(Fem*MCAT)`.)

```
> model5 <- glm(Acceptance ~ GPA + MCAT + Fem + Fem:MCAT, family=binomial)
> summary(model5)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8644  -0.9001   0.2219   0.6508   2.1709

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -34.4369    11.3066  -3.046  0.00232 **
GPA           5.8178     2.0661   2.816  0.00486 **
MCAT          0.3661     0.1937   1.890  0.05880 .
Fem          12.8878     8.9405   1.442  0.14944
MCAT:Fem     -0.3214     0.2447  -1.313  0.18905
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

```
(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 75.791  on 54  degrees of freedom
Residual deviance: 48.849  on 50  degrees of freedom
AIC: 58.849
```

10.5 Case Study: Bird Nests

The modeling in the bird nests study should be fairly straightforward applications of the type of code explained earlier in the chapter, but we will show here the R code for producing the boxplots of Figure 10.18. The command `par(mfrow=c(1,2))` sets up a 1 by 2 panel for the 2 separate boxplots to go into. In the calls to the `boxplot` function, we use a model statement as the input, rather than the pair of vectors option. So, for example, `Length` $\sim$ `Closed` creates a model that splits the Lengths into two groups, based on the values (0 or 1) of Closed; exactly what we want. Notice the use of the `main` argument to add headings to the graphs.

```
> par(mfrow=c(1,2))  # set up a two-panel window for two boxplots
> attach(BirdNest.df) # attach the dataset

> boxplot(Length~Closed,main="Length")
> boxplot(TotCare~Closed,main="TotCare")
```

CHAPTER 11

Logistic Regression: Additional Topics

This chapter of the *Companion* describes R procedure for handling the additional topics of logistic regression covered in Chapter 11. We again let the textbook's examples drive the exposition in this Companion. The first section of the chapter, Section 11.1 does not involve any computation, so we go on to the next section.

11.2 Assessing Logistic Regression Models

Example 11.2: Putting

The code below reproduces the calculations for this putting example. We include R code for constructing a table that includes empirical and estimated proportions and empirical and estimated logits. We also include R code to fit the linear logistic and the saturated models to these data. Keep in mind that we need to enter the data into the `glm` function as binomial data, that is, as counts of makes and misses.

After attaching the data frame, we recreate a table of makes and misses, which we call `a`. From this we create a table of column proportions, which we name `col.props` and then rename `b` after rounding proportions to 3 places. The line that defines `col.props` uses the R function `prop.table` where the `2` argument refers to columns; an argument value of `1` would give row proportions.

We next define a vector of empirical logits, which we denote by `c`. The code
`d <- tapply(Putts.glm1$fitted,Length,unique)` bears explanation. The `tapply` function takes the vector `Putts.glm1$fitted` of fitted values from the linear logistic model, uses the `Length` variable as a "by" variable, and performs the `unique function` for each value of `Length`. The `unique` function takes a vector as input and returns a vector consisting of each unique value listed just once. For example, `unique(c(1,2,2,3,3,3))` would return a value of `1 2 3`. But since for each value of `Length` we know there is only one fitted value, this line of code returns a single fitted value for each length of putt. We complete this section of code by defining the table `table11.2` using the `rbind` function, which combines a set of tables or vectors row-wise.

```
> attach(Putts1.df)
> names(Putts1.df)
[1] "Length" "Made"

> a <- table(Made,Length)[c(2,1),] #a is the table with rows reversed
> a
    Length
Made  3  4  5  6  7
   1 84 88 61 61 44
   0 17 31 47 64 90

> col.props <- prop.table(a,2)  # compute column proportions
> col.props
    Length
Made         3         4         5     6         7
   1 0.8316832 0.7394958 0.5648148 0.488 0.3283582
   0 0.1683168 0.2605042 0.4351852 0.512 0.6716418

> b <- round(col.props,3) # Round column proportions, put into b
> b
    Length
Made     3     4     5     6     7
   1 0.832 0.739 0.565 0.488 0.328
   0 0.168 0.261 0.435 0.512 0.672
> c <- log(b[1,]/(1-b[1,]))  # Compute column-wise logits
> c
          3           4           5           6           7
 1.59986846  1.04077751  0.26147970 -0.04800922 -0.71724473

> round(c,2)  # Round logits
    3     4     5     6     7
 1.60  1.04  0.26 -0.05 -0.72
```

```
# For each Length, the fitted value is identical, so
# the unique function gives this value for each Length.
> d <- tapply(Putts.glm1$fitted,Length,unique)
> d
        3         4         5         6         7
0.8261256 0.7295364 0.6049492 0.4650541 0.3304493

> round(d,3)  # Round to 3 places, as in text
    3     4     5     6     7
0.826 0.730 0.605 0.465 0.330

> e <- log(d/(1-d))
> e
         3          4          5          6          7
 1.5584133  0.9922716  0.4261300 -0.1400117 -0.7061534

> round(e,3)
     3      4      5      6      7
 1.558  0.992  0.426 -0.140 -0.706

# Now, reproduce Table 11.2:

> table11.2 <- rbind(b[1,],round(c,2),round(d,3),round(e,3))

> row.names(table11.2) <- c("Saturated Prob Est","Logits of Satur Est",
                            "Prob Est. (Model)","Logit Est. (Model)")
> table11.2
                         3     4     5      6      7
Saturated Prob Est   0.832 0.739 0.565  0.488  0.328
Logits of Satur Est  1.600 1.040 0.260 -0.050 -0.720
Prob Est. (Model)    0.826 0.730 0.605  0.465  0.330
Logit Est. (Model)   1.558 0.992 0.426 -0.140 -0.706
```

Now we reproduce the logistic model given in the text, where we enter the data as binomial counts, and model putt success against putt length. The code is below, and we note that the output agrees with that in the text.

```
> a
    Length
Made  3  4  5  6  7
   1 84 88 61 61 44
   0 17 31 47 64 90
> Makes <- a[1,]
```

```
> Misses <- a[2,]
> Lengths <- 3:7
> model1 <- glm(cbind(Makes,Misses) ~ Lengths, family=binomial)
> summary(model1)
Deviance Residuals:
       3         4         5         6         7
 0.14800   0.24555  -0.84871   0.51388  -0.05149

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.25684    0.36893   8.828   <2e-16 ***
Lengths     -0.56614    0.06747  -8.391   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 81.3865  on 4  degrees of freedom
Residual deviance:  1.0692  on 3  degrees of freedom
AIC: 30.175
```

Next, we reproduce the saturated model by treating each putt length as a separate factor and thus fitting a separate probability of success for each length. We see the residual deviances from these two chunks of R output that allow us to compute the "drop in deviance" test explained in the text.

```
> model2 <- glm(cbind(Makes, Misses) ~ factor(Lengths), family=binomial)
> summary(model2)
Coefficients:
                 Estimate Std. Error z value Pr(>|z|)
(Intercept)        1.5976     0.2659   6.007 1.89e-09 ***
factor(Lengths)4  -0.5543     0.3382  -1.639    0.101
factor(Lengths)5  -1.3369     0.3292  -4.061 4.90e-05 ***
factor(Lengths)6  -1.6456     0.3205  -5.134 2.84e-07 ***
factor(Lengths)7  -2.3132     0.3234  -7.154 8.46e-13 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 8.1387e+01  on 4  degrees of freedom
Residual deviance: 5.3291e-15  on 0  degrees of freedom
AIC: 35.106
```

Example 11.4: Another putting dataset

After reading in and attaching the `Putts2.df` dataset, we fit the logistic models whose R code is given in the text. Recall that overdispersion is the issue here. The code below fits the model using the `family=binomial` argument, which replicates the overdispersion example of the text. This we follow with the refit of the model using the `family=quasibinomial`, which gives the model that adjusts the fit for the overdispersion. Since the output in the text was produced by R code, it is not surprising that our output agrees with the text.

```
> Putts2.df <- read.csv(file=file.choose())
> names(Putts2.df)
[1] "Length" "Made"   "Missed" "Trials"

> Makes <- c(79,94,60,65,40)
> Misses <- c(22,25,48,60,94)
> Putts3.df <-data.frame(Makes, Misses)
> names(Putts3.df)

> attach(Putts3.df)

> Putts.glm3 <- glm(cbind(Makes,Misses) ~ Lengths, family=binomial)
> summary(Putts.glm3)
Deviance Residuals:
     1       2       3       4       5
-1.131   1.522  -1.043   1.230  -0.793

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.25684    0.36893   8.828   <2e-16 ***
Lengths     -0.56614    0.06747  -8.391   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 87.1429  on 4  degrees of freedom
Residual deviance:  6.8257  on 3  degrees of freedom
AIC: 35.93

> Putts.glm4 <- glm(cbind(Makes,Misses) ~ Lengths, family=quasibinomial)
> summary(Putts.glm4)
```

```
Deviance Residuals:
     1       2       3       4       5
-1.131   1.522  -1.043   1.230  -0.793

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.2568     0.5552   5.866  0.00988 **
Lengths      -0.5661     0.1015  -5.576  0.01139 *
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for quasibinomial family taken to be 2.264714)

    Null deviance: 87.1429  on 4  degrees of freedom
Residual deviance:  6.8257  on 3  degrees of freedom
```

Residual Diagnostics: Assessing the Conditions

We begin with Example 11.5, which refers us back to the model we called `CAFE.glm`, a logistic model of `Vote` on `LogContr` and `Dem`. We include the summary output below.

```
Call:
glm(formula = Vote ~ LogContr + Dem, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.4117  -0.6570   0.3600   0.5622   2.3625

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -6.8402     2.4903  -2.747 0.006020 **
LogContr      2.1659     0.6131   3.533 0.000411 ***
Dem          -1.7328     0.5804  -2.985 0.002833 **
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 132.813  on 99  degrees of freedom
Residual deviance:  87.336  on 97  degrees of freedom
AIC: 93.36
```

We now give the code that produces the residual plot using Pearson residuals and lets us identify outlying points by the Senator involved.

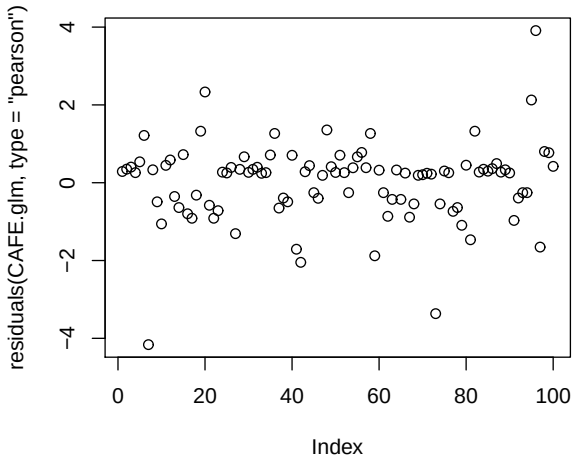


Figure 11.1: *CAFE.glm model residuals with two outliers labeled*

```
attach(CAFE.df)
plot(residuals(CAFE.glm,type="pearson"))
identify(residuals(CAFE.glm,type="pearson"),label=Senator)
```

Assessing Prediction

Table 11.5 is worth reproducing, since the graphic is such a simple way of assessing the predictive success of a fitted model. We give the R code below, with this explanation. First, we refit the model `Med.glm3`; if you saved it from earlier, you could dispense with these lines of code. Then we define the vector `Accept.hat`, or predicted acceptances to medical school based upon an estimated probability of success greater than .5. We then form the 2×2 Table 11.5 using the `table` function. Note the agreement with the text. Finally, we rename the row headings to make the table a bit more transparent.

```
# Re-fit a model discussed earlier.
> attach(MedGPA.df)
> Med.glm3 <- glm(Acceptance ~ MCAT + GPA10, family=binomial)
> summary(Med.glm3) # We suppress the output, which we have seen earlier.

> Accept.hat <- Med.glm3$fitted > .5  # Define predicted acceptances
> attach(MedGPA.df)
> table(Acceptance,Accept.hat) # Obtain prediction assessment table
          Accept.hat           # Note agreement with text
```

```
Acceptance FALSE TRUE
         0    18    7
         1     7   23

# Make the output more transparent.
> tab <- table(Acceptance,Accept.hat)
> row.names(tab)
[1] "0" "1"
> row.names(tab) <- c("Deny","Accept")
> tab
          Accept.hat
Acceptance FALSE TRUE
    Deny      18    7
    Accept     7   23
```

11.3 Randomization Tests

We describe now the R code for Example 11.6. The first randomization test uses the binary `Sex` variable as the predictor. The table shows that the males were more likely to improve, with an odds ratio of 3 = (9*2)/(6*1). This will be the benchmark odds ratio against which our randomly created datasets will be compared.

We begin by defining the two variables and creating the table `arch.tab`. We define then an odds ratio function, `or`, and verify the odds ratio of 3. The randomization part follows. Since we have not talked much about writing programs in the `Companion` we will take this a bit more slowly than usual.

First, we put the odds ratio for the original data into a variable called `TestStat0`. The `n.samples <- 100000` line says we are going to do a repetitive task 100000 times. It is always a good idea, when writing such code, to start out with a really small value for the number of iterations. This is not complicated code, but we still proceeded cautiously and used `n.samples <- 10` as we tried out the work. The book explains the idea, but in the spirit of "repetition is the salt and pepper of life," we will say it again, and slowly. The code between the two comments about begin and end of the loop is called a "for loop." The line `for (i in 1:n.samples)` says that all the code between the pair of curly braces that follows will be executed `n.samples` times (in our case 100000).

Each time through the code something changes and something is recorded. Here is what changes. Imagine the original data set of the 40 values of `improve`, a vector here of 40 0s and 1s, alongside (column-wise, you may imagine) the 40 values of the `female` vector of 0s and 1s. The null hypothesis says the two variables are unrelated to one another, so within each iteration we permute completely at random the 40 `female` values, as if they were meaningless labels (which is what the null says they are). With any given permutation, you can imagine a new data set: the 40 `improve` values in the same order, but the 40 `female` values in a new and random order. Now, re-form the

2×2 table of counts and compute the odds ratio again. Put this odds ratio into a vector of length `n.samples` (100000, in our case) that we decide to call `TestStat`.

This iterative process of randomly generated odds ratios is repeated `n.samples` times. At the end of this process `TestStat` should have `n.samples` values of which our original odds ratio, `TestStat0` should look kind of middle-of-the-pack if the null really is true. Thus, we compute the p-value by counting out how many values of `TestStat` are greater than or equal to `TestStat0`, as always with a p-value, getting a measure of how extreme our data look; small p-value, more extreme. The p-value we get is somewhat more significant than found in the text, namely .4091. Still, we would deem the relationship here to be clearly nonsignificant.

```
# assume the data are in data frame archery.df
> female <- as.numeric(archery.df$=="f") # indicator for female
> improve <- as.numeric(archery.df$Improvement > 0)
  #indicator for a positive improvement

> arch.tab
       female
improve FALSE TRUE
  FALSE     1    2
  TRUE      9    6

> or <- function(x) (x[1,2]*x[2,1])/(x[2,2]*x[1,1])
> or(arch.tab)
[1] 3

dataset <- data.frame(improve,female)
attach(dataset)  # attach dataset so we can refer easily
                 # to its variables.

# First we compute and record at TestStat0 the odds ratio
# test statistic for the original two-way table (the 3):
TestStat0 <- or(arch.tab)

# We set n.samples, which is the number of random permutations
# we choose to use in our randomization test.
n.samples <- 100000

# We create a vector of length n.samples that initially contains
# NA values, but will get filled up with test statistic values
# as the simulation progresses.
TestStat <- rep(NA, n.samples)
```

```
# Now here is the simulation that performs the repeated
# re-arrangements of the Peaceworks values.
# Each time through the loop a new two-way
# table is created, from which we get a new
# odds ratio test statistic to add to the vector
# TestStat.

# Loop starts here
for (i in 1:n.samples){
     new.female <- sample(female)
     tab <- table(new.female, improve)
     TestStat[i] <- or(tab)
}
# End of Loop

> p.value <- sum(TestStat >= TestStat0)/length(TestStat)
> p.value
[1] 0.4091
```

We now look at the second randomization test from the text relative to the archery study, this one involving the same `improve` variable for the response, but now with the quantitative explanatory variable, `attendance`. The code for this test follows, and it is dangerously similar to that we just discussed.

The first difference is in the definition of the `or` function. Here the test statistic is the odds ratio computed as the exponentiation of the slope parameter from the logistic model. The observed value, which we store in `TestStat0`, is about .83, meaning the odds of improving decrease by a factor of .83 for each unit increase in attendance. (This would be alarming were it not for what the significance test reveals momentarily.) In our randomly generated vector of odds ratios, we now are interested in those less than or the same as this .83. The fraction of these extreme values is the p-value, and it is large enough to be nowhere near statistical significance, namely 0.5147.

```
# assume the data are in data frame archery.df
> attendance <- archery.df$Sex=="f" # attendance is a quantitative predictor
> improve <- as.numeric(archery.df$Improvement > 0)  #indicator for a positive improvement

# We replicate the book's logistic fit.
> model <- glm(improve ~ attendance, family=binomial)
> summary(model)
```

```
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   5.5792    11.0819   0.503    0.615
attendance   -0.1904     0.5278  -0.361    0.718

# Notice the slope coefficient is stored in model.
> model$coef[2]
attendance
-0.1904289

# A simple way to get the odds ratio;
# the text rounds it to .83.

> exp(model$coef[2])
attendance
 0.8266045

# So now define the odds ratio from the logistic output.
> or <- function(x) exp(glm(improve ~ x, family=binomial)$coef[2])
> or(attendance)
        x
0.8266045   # agrees with earlier result

dataset <- data.frame(improve,attendance)
attach(dataset)  # attach dataset so we can refer easily
                 # to its variables.

# First we compute and record at TestStat0 the odds ratio
# test statistic for the original two-way table (the 3):
TestStat0 <- or(attendance)

# We set n.samples, which is the number of random permutations
# we choose to use in our randomization test.
n.samples <- 10000

# We create a vector of length n.samples that initially contains
# NA values, but will get filled up with test statistic values
# as the simulation progresses.
TestStat <- rep(NA, n.samples)

# Now here is the simulation that performs the repeated
# rearrangements of the Peaceworks values.
```

```
# Each time through the loop a new two-way
# table is created, from which we get a new
# odds ratio test statistic to add to the vector
# TestStat.

# Loop starts here
for (i in 1:n.samples){
     new.attendance <- sample(attendance)
     TestStat[i] <- or(new.attendance)
}
# End of Loop

> p.value <- sum(TestStat <= TestStat0)/length(TestStat)
> p.value
[1] 0.5147
```

11.4 Analyzing Two-way Tables with Logistic Regression

Example 11.7: Joking for a tip

To reproduce Figure 11.8 we use the R code below. We want to create a data frame that could have produced the two-way table for the Joke experiment. We begin by creating a numeric data frame called `data` that sets up a data frame of proper dimensions. R creates default names for the variables, but we reassign them values of `Response` and `Condition` in that third line of code. Then, we create the contingency table of counts using the `table` command. Finally, we use a new R function `barplot` to create the desired barplot.

The `barplot` function contains a `ylim=c(0,90)` argument that creates a bit more space in the y direction to allow for a proper legend. We use `legend` to produce the legend, the `locator(1)` argument allowing the user to click in the graph to place the legend. This step can take some trial-and-error, and it was here that we decided to use `cex=.7` to reduce the font size a bit. The `c("No Tip","Tip"),fill=c("white","black")` arguments align the tipping conditions to the shading in the bars.

```
> data <- data.frame(numeric(211),ncol=2)  # Create a data frame
                                           # of all 0s of proper dimensions.
> names(data) <- c("Condition","Response") # Create better variable names.
```

```
> data$Response <- factor(c(rep("Tip",30),rep("NoTip",42),rep("Tip",14),
+ rep("NoTip",60), rep("Tip",16),rep("NoTip",49)))
#Now make the data frame contain
#the proper categorical data

> data$Condition <- factor(c(rep("Joke",72),rep("Ad",74),rep("none",65)))

# Create the contingency table:
> Joke.tab <- table(data$Response,data$Condition)[c(2,1),c(2,1,3)]
> Joke.tab

       Joke Ad none
  Tip    30 14   16
  NoTip  42 60   49


# Produce the barplot, with a legend
> barplot(Joke.tab,ylim=c(0,90))
> legend(locator(1),c("No Tip","Tip"),fill=c("white","black"),cex=.7)
```

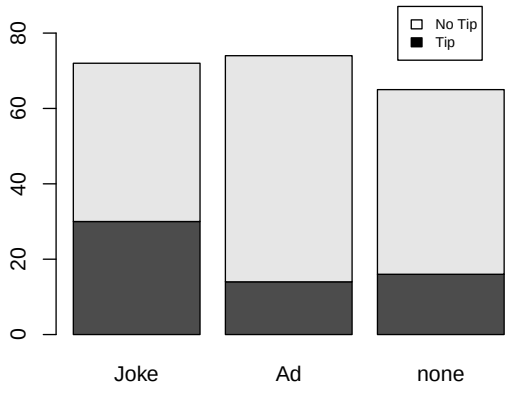


Figure 11.2: *Proportion of tips within each card group*

Logistic Regression Results for the 2×3 Table

We now add the logistic model fit and the chi-square analysis to replicate the textbook's results. To obtain the logistic model, use the code below. First, we read the data in. Notice that the indicator variables for `Joke`, `Ad`, and `None` are already defined in the dataset. Then, we fit the logistic model `TipJoke.glm` in the usual way. We follow this with calculations of odds ratios (the vector `or`) and confidence intervals for the odds ratios (the matrix `ci`), and again we note that the confidence intervals differ slightly from the Minitab intervals given in the text. We use `cbind` to combine them into a table similar to that in the text.

```
> TipJoke.df <- read.csv(file=file.choose())
> names(TipJoke.df)
[1] "Card" "Tip"  "Ad"   "Joke" "None"

> TipJoke.glm <- glm(Tip ~ Joke + Ad, family=binomial)
> summary(TipJoke.glm)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.0383  -0.7518  -0.6476   1.3232   1.8248

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -1.1192     0.2879  -3.887 0.000101 ***
Joke          0.7828     0.3742   2.092 0.036471 *
Ad           -0.3361     0.4135  -0.813 0.416413
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 251.94  on 210  degrees of freedom
Residual deviance: 242.14  on 208  degrees of freedom
AIC: 248.14

# Obtain the odds ratios:
> or <- round(exp(TipJoke.glm$coef)[c(2,3)],2)
Joke   Ad
2.19 0.71

# Now obtain confidence intervals:
> ci <- round(exp(confint(TipJoke.glm)),2)[c(2,3),]
> ci
```

```
     2.5 % 97.5 %
Joke  1.06   4.63
Ad    0.31   1.61

# Combine them into a single table:
> cbind(or,ci)
       or 2.5 % 97.5 %
Joke 2.19  1.06   4.63
Ad   0.71  0.31   1.61
```

Chi-square Test for a $2 \times k$ Table

We now add the results of the chi-square test for contingency tables using R's `chisq.test` function. By assigning the output of this function call to a variable, `Tip.chisq`, we can easily extract the expected values and the p-value.

```
# Joke.tab is a table of observed counts:
> Joke.tab <- table(Tip,Card)
> Joke.tab
   Card
Tip Ad Joke None
  0 60   42   49
  1 14   30   16


# The object Tip.chisq are results of a chi-square test,
# from which we extract expected values, rounded to two places:

> Tip.chisq <- chisq.test(Joke.tab)
> round(Tip.chisq$expected,2)
   Card
Tip    Ad  Joke  None
  0 52.96 51.53 46.52
  1 21.04 20.47 18.48

> Tip.chisq

        Pearson's Chi-squared test

data:  Joke.tab
X-squared = 9.9533, df = 2, p-value = 0.006897
```

The Two-Sample Test and the Chi-square Test for a 2×2 Table (review)

Finally, we conclude our *R Companion* by giving code to produce two procedures you may have learned about in your STAT1 course. For this, we use the two-by-two table from the zapping migraines example. Recall that in the table shown below, the `zap.tab`, columns represent a TMS treatment versus a placebo and the outcomes are either that the patient is or is not pain free after a period of time.

The code below first uses the `prop.test` function to calculate the z-test. Actually, it performs the chi-square test, which is equivalent to the z-test when the two-sided alternative is the goal. Note the inclusion of a 95% confidence interval for the difference in proportions. We run the function again,this time using the `alternative=‘‘greater")` argument, which gives the one-sided test. In a situation where our interests and expectations were that the TMS treatment would, if anything, have a positive effect, it would probably be natural in this context to want the one-sided. Here we get a one-side confidence interval.

Then we run the chi-square test. Note the p-value is identical for that of the z-test.

```
> zap.tab
              TMS Placebo
Pain Free      39      22
Not Pain Free  61      78

> prop.test(zap.tab)

        2-sample test for equality of proportions with continuity correction

data:  zap.tab
X-squared = 6.0384, df = 1, p-value = 0.01400
alternative hypothesis: two.sided
95 percent confidence interval:
 0.04266497 0.35832571
sample estimates:
   prop 1    prop 2
0.6393443 0.4388489


> prop.test(zap.tab, alternative="greater")

        2-sample test for equality of proportions with continuity correction

data:  zap.tab
X-squared = 6.0384, df = 1, p-value = 0.006999
```

```
alternative hypothesis: greater
95 percent confidence interval:
 0.06614378 1.00000000
sample estimates:
   prop 1    prop 2
0.6393443 0.4388489

> chisq.test(zap.tab)

        Pearson's Chi-squared test with Yates' continuity correction

data:  zap.tab
X-squared = 6.0384, df = 1, p-value = 0.01400
```

Index