Appendix A

Computing in R

Computing is central to our conception of statistics, both in practice—we can do many more things in statistics if we do not restrict ourselves to problems with analytical solutions—and in teaching, as we use simulation and graphics to understand data, models, and the statistical properties of model fits. In this appendix, we explain how to get started with R and Stan and how to access the data and examples used in the book. We then give a brief introduction to R. There are many R tutorials online that can provide additional information; the introduction here is focused on some of the data processing, analysis, and programming tasks that arise in applied regression. We also provide pointers to useful packages for data analysis in R and Stan.

A.1 Downloading and installing R and Stan

We do our computing in the open-source package R, a command-based statistical software environment that you can download and operate on your own computer, using RStudio, a free graphical user interface for R. We fit Bayesian regressions in `rstanarm`, an R package that calls Stan, an open-source program written in C++. The `rstanarm` package includes a library of already compiled Stan models for fitting linear and logistic regression, generalized linear models, and a selection of other models that can be called using `stan_glm` and other functions. See `www.mc-stan.org/rstanarm` for documentation and vignettes for modeling examples. Stan itself is more general, allowing the user to program and fit arbitrary Bayesian models.

R and RStudio

To set up R, go to `www.r-project.org` and click on the "download R" link. This will take you to a list of mirror sites. Choose any of these. Now click on the link under "Download and Install R" at the top of the page for your operating system (Linux, Mac, or Windows) and download the binaries corresponding to your system. Follow all the instructions. Default settings should be fine.

Go to `www.rstudio.com`, click on "Download RStudio," click on "Download RStudio Desktop," and click on the installer for your platform (Windows, Mac, etc.). Then a download will occur.

When the download is done, click on the RStudio icon to start an R session.

To check that R is working, go to the Console window within RStudio. There should be a ">" prompt. Type `2+5` and hit Enter. You should get `[1] 7`. From now on, when we say "type" `_____`, we mean type `_____` into the R console and hit Enter.

You will also want to install some R *packages*, in particular, `ggplot2` (for building graphs), `knitr` (which allows you to process certain R documentation), `bayesplot` (which has convenient functions for displaying fitted models), `rstanarm` (which has functions for fitting Bayesian regression models using simulation), `rstan` (which allows you to fit more general models in Stan), `loo` (fast leave-one-out cross validation; see Section 11.8), `survey` (which contains functions for classical survey analysis, some of which we use in working with propensity scores in Chapter 20), `arm` (which includes functions we have written for applied regression modeling, some of which are used in

Chapter 21), and `rprojroot` (which has been used in the code for this book to make it easier to work with many folders). Some of the example code also uses other packages which you can see at the beginning of each code file. You can install these packages from RStudio by clicking on the Tools tab and then on Install Packages and then entering the names of the packages and clicking Install. When you want to use a package, you load it as needed, for example by typing `library("brms")`.

Stan and rstanarm

If your only use for Stan is to fit the regression models in this book, it will be enough to install `rstanarm` as discussed above, and then begin any R session with,

```
library("rstanarm")
```

We almost always do our computing by writing scripts, rather than simply typing commands into the R console window; thus, we recommend simply typing the above line at the beginning of any of your R scripts that include calls to `stan_glm`.

For more flexible Bayesian modeling you will need to use Stan itself, for which downloads, documentation, and other information are available at `www.mc-stan.org`. Follow the instructions to set up `rstan`, the R interface to Stan. This will automatically install Stan itself on your computer, which in turn requires a C++ compiler; again, the necessary instructions are at the `rstan` set-up page.

Examples to get started are in the `rstan` and `rstanarm` documentation. We list some useful functions in Section A.8.

A.2 Accessing data and code for the examples in the book

The data and code used in the examples and to make the figures in this book are available on Github, linked from `www.stat.columbia.edu/~gelman/regression`. You can copy all the files to your own computer by clicking on the Clone or Download button and then clicking on Download ZIP, or by using a direct link. If you are familiar with `git` version control, you can also clone the repository. For quick viewing of code and example results, they are also available as web pages at this directory.

A.3 The basics

Computing is central to modern statistics at all levels, from basic to advanced. If you already know how to program, great. If not, consider this a start.

Try a few things, typing these one line at a time and looking at the results on the console:

```
1/3
sqrt(2)
curve(x^2 + 5, from=-2, to=2)
```

These will return `0.3333333`, `1.414214`, and a new graphics window plotting the curve $y = x^2 + 5$.

Finally, quit your R session by closing the RStudio window.

Calling functions and getting help

Reopen R and play around, using the assignment function (“<-”). To start, type the following lines into your script.R file and copy-and-paste them into the R window:

```
a <- 3
print(a)
b <- 10
a + b
a*b
```

```
exp(a)
10^a
log(b)
log10(b)
a^b
round(3.435, 0)
round(3.435, 1)
round(3.435, 2)
```

R is based on *functions*, which include mathematical operations (`exp`, `log`, `sqrt`, and so forth) and lots of other routines (`print`, `round`, . . .).

The function `c()` concatenates numbers together into a vector. For example, type `c(4,10,-1,2.4)` in the R console or type the following:

```
x <- c(4,10,-1,2.4)
print(x)
```

The function `seq` creates an equally-spaced sequence of numbers; for example, `seq(4,54,10)` returns the sequence, 4, 14, 24, 34, 44, 54. The `seq` function works with non-integers as well: try `seq(0,1,0.1)` or `seq(2,-5,-0.4)`. For integers, `a:b` is shorthand for `seq(a,b,1)` if `b>a`, or `seq(a,b,-1)` if `b<a`. Let's try a few more commands:

```
c(1, 3, 5)
1:5
c(1:5, 1, 3, 5)
c(1:5, 10:20)
seq(-1, 9, 2)
```

You can get help on any function using "?" in R. For example, type `?seq`. This should open a window with a help file for `seq`. R help files typically have more information than you'll know what to do with, but if you scroll to the bottom of the page you'll find some examples that you can cut and paste into your console. Whenever you are trying out a new function, we recommend using "?" to view the help file and running the examples at the bottom to see what happens.

Sampling and random numbers

Here's how to get a random number, uniformly distributed between 0 and 100:

```
runif(1, 0, 100)
```

And now 50 more random numbers:

```
runif(50, 0, 100)
```

Suppose we want to pick one of three colors with equal probability:

```
color <- c("blue", "red", "green")
sample(color, 1)
```

Suppose we want to sample with unequal probabilities:

```
color <- c("blue", "red", "green")
p <- c(0.5, 0.3, 0.2)
sample(color, 1, prob=p)
```

Or we can do it all in one line, which is more compact but less readable:

```
sample(c("blue","red","green"), 1, prob=c(0.5,0.3,0.2))
```

Data types

Numeric data. In R, numbers are stored as *numeric* data. This includes many of the examples above as well as special constants such as `pi`.

Big and small numbers. R recognizes scientific notation. A million can be typed in as `1000000` or `1e6`, but not as `1,000,000`. (R is particular about certain things. Capitalization matters, "`,`" doesn't belong in numbers, and spaces usually aren't important.) Scientific notation also works for small numbers: `1e-6` is `0.000001` and `4.3e-6` is `0.0000043`.

Infinity. Type these into R, one line at a time, and see what happens:

```
 1/0
-1/0
 exp(1000)
 exp(-1000)
 1/Inf
 Inf + Inf
-Inf - Inf
 0/0
 Inf - Inf
```

Those last two operations return `NaN` (Not a Number); type `?Inf` for more on the topic. In general we try to avoid working with infinity, but it is convenient to have `Inf` for those times when we accidentally divide by 0 or perform some other illegal mathematical operation.

Missing data. In R, `NA` is a special keyword that represents missing data. For more information, type `?NA`. Try these commands in R:

```
NA
2 + NA
NA - NA
NA / NA
NA * NA
c(NA, NA, NA)
c(1, NA, 3)
10 * c(1, NA, 3)
NA / 0
NA + Inf
is.na(NA)
is.na(c(1, NA, 3))
```

The `is.na` function tests whether or not the argument is `NA`. The last line operates on each element of the vector and returns a vector with three values that indicate whether the corresponding input is NA.

Character strings. Let's sample a random color and a random number and put them together:

```
color <- sample(c("blue","red","green"), 1, prob=c(0.5,0.3,0.2))
number <- runif(1, 0, 100)
paste(color, number)
```

Here's something prettier:

```
paste(color, round(number,0))
```

TRUE, FALSE, and ifelse

Try typing these:

```
2 + 3 == 4
2 + 3 == 5
1 < 2
2 < 1
```

In R, the expressions `==`, `<`, `>` are *comparisons* and return a logical value, `TRUE` or `FALSE` as appropriate. Other comparisons include <= (less than or equal), >= (greater than or equal), and `!=` (not equal).

Comparisons can be used in combination with the `ifelse` function. The first argument takes a logical statement, the second argument is an expression to be evaluated if the statement is true, and the third argument is evaluated if the statement is false. Suppose we want to pick a random number between 0 and 100 and then choose the color red if the number is below 30 or blue otherwise:

```
number <- runif(1, 0, 100)
color <- ifelse(number<30, "red", "blue")
```

Loops

A key aspect of computer programming is *looping*—that is, setting up a series of commands to be performed over and over. Start by trying out the simplest possible loop:

```
for (i in 1:10){
  print("hello")
}
```

Or:

```
for (i in 1:10){
  print(i)
}
```

Or:

```
for (i in 1:10){
  print(paste("hello", i))
}
```

The curly braces define what is repeated in the loop. The spaces and line breaks are not necessary—one could just as well do `for(i in 1:10)print(paste("hello",i))`—but they improve readability.

Here's a loop of random colors:

```
for (i in 1:10){
  number <- runif(1, 0, 100)
  color <- ifelse(number<30, "red", "blue")
  print(color)
}
```

Working with vectors

In R, a vector is a list of items. These items can include numerics, characters, or logicals. A single value is actually represented as a vector with one element. Here are some vectors:

- (1, 2, 3, 4, 5)
- (3, 4, 1, 1, 1)
- ("A", "B", "C")

Here's the R code to create these:

```
x <- 1:5
y <- c(3, 4, 1, 1, 1)
z <- c("A", "B", "C")
```

And here's a random vector of 5 random numbers between 0 and 100:

```
u <- runif(5, 0, 100)
```

Mathematical operations on vectors are done componentwise. Take a look:

```
x
y
x + y
1000*x + u
```

There are scalar operations on vectors:

```
1 + x
2 * x
x / 3
x^4
```

We can summarize vectors in various ways, including the sum and the average (called the "mean" in statistics jargon):

```
sum(x)
mean(x)
```

We can also compute weighted averages if we know the weights. We illustrate with a vector of three elements:

```
x <- c(100, 200, 600)
w1 <- c(1/3, 1/3, 1/3)
w2 <- c(0.5, 0.2, 0.3)
```

In the above code, the vector of weights `w1` has the effect of counting each of the three items equally; vector `w2` counts the first item more. Here are the weighted averages:

```
sum(w1*x)
sum(w2*x)
```

Or suppose we want to weight in proportion to population:

```
N <- c(310e6, 112e6, 34e6)
sum(N*x)/sum(N)
```

Or, equivalently,

```
N <- c(310e6, 112e6, 34e6)
w <- N/sum(N)
sum(w*x)
```

The `cumsum` function does the cumulative sum. Try this:

```
a <- c(1, 1, 1, 1, 1)
cumsum(a)
a <- c(2, 4, 6, 8, 10)
cumsum(a)
```

Subscripting

Vectors can be indexed by using brackets, "`[ ]`". Within the brackets we can put in a vector of elements we are interested in either as a vector of numbers or a logical vector. When using a vector of numbers, the vector can be of arbitrary length, but when indexing using a logical vector, the length of the vector must match the length of the vector you are indexing. Try these:

```
a <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J")
a[1]
a[2]
a[4:6]
a[c(1,3,5)]
a[c(8,1:3,2)]
a[c(FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)]
```

As we have seen in some of the previous examples, we can perform mathematical operations on vectors. These vectors have to be the same length, however. If the vectors are not the same length, we can subset the vectors so they are compatible. Try these:

```
x <- c(1, 1, 1, 2, 2)
y <- c(2, 4, 6)
x[1:3] + y
x[3:5] * y
y[3]^x[4]
x + y
```

The last line runs but produces a warning. These warnings should not be ignored since it isn't guaranteed that R would carry out the operation as you intended.

A.4 Reading, writing, and looking at data

Your working directory

Choose a *working directory* on your computer where you will do your R work. Suppose your working directory is `c:/myfiles/stat/`. Then you should put all your data files in this directory, and all the files and graphs you save in R will appear here too. To set your working directory in RStudio, click on the Session tab and then on Set Working Directory and then on Choose Directory and then navigate from there.

RStudio has several subwindows: a text editor, the R console, a graphics window, and a help window. It's generally best to type your commands into the RStudio text editor, select the lines you want to run, and run them by pressing Ctrl-Enter. When your session is over, you can save the contents of the text editor into a plain-text file with a name such as `todays_work.R` that you can save in your working directory.

Now go to the R console and type `getwd()`. This shows your R working directory, which you can change by typing `setwd("c:/myfiles/stat/")`, or whatever you would like to use. In RStudio you can also choose the working directory using menu `Session -> Set Working Directory`. You can type `getwd()` again to confirm that you are in the right place.

The code for the book uses the `rprojroot` package, which makes it easy to run code from different subfolders without need to change the working directory to a specific folder. It is sufficient to set the working directory to the main demo folder or any of the subfolders.

Reading data

Let's read some data into R. The file `heads.csv` has data from a coin-flipping experiment done in a previous class. Each student flipped a coin 10 times, and we have a count of the number of students who saw exactly 0, 1, 2, . . . , 10 heads. The data file is in the folder `Coins` at our data and code repository. Start by going to this location, finding the directory, downloading the file, and saving it as `heads.csv` in your working directory (for example, `c:/myfiles/stat/`).

Now read the file into R:

```
heads <- read.csv("heads.csv")
```

Typing the name of any object in R displays it. So type heads and look at what comes out.

If you have tabular data separated by spaces or tabs, you can just use the `read.table` function. Here is an example of the file `mile.txt` from the folder `Mile`:

```
mile <- read.table("mile.txt", header=TRUE)
mile[1:5,]
```

The `header=TRUE` argument is appropriate here because the first line of the file mile.txt is a "header," that is, a list of column names. If the file had just data with no header, we would simply call `read.table("mile.txt")` with no header argument.

Writing data

You can save data into a file using `write` instead of `read`. For example, to write the R object heads into a comma-separated file `output1.csv`, we would type `write.csv(heads,"output1.csv")`. To write it into a space-separated file `output2.txt`, it's just `write.table(heads,"output2.txt")`.

Examining data

At this point, you should have two variables in your R environment, heads and `mile`. You can see what you have in your session by clicking on the Environment tab in the RStudio window.

Data frames, vectors, and subscripting. Most of the functions used to read data return a structure called a *data frame*. You can see an example by typing `class(heads)`. Each column of a data frame is a vector. We can access the first column of heads by typing `heads[,1]`. Data frames are indexed using two vectors inside "`[`" and "`]`"; the two vectors are separated by a comma. The first vector indicates which rows you are interested in, and the second vector indicates what columns you are interested in. For example, `heads[6,1]` shows the number of heads observed and `heads[6,2]` shows the number of students that observed that number of heads. Leaving it blank is shorthand for including all. Try:

```
heads[6,]
heads[1:3,]
heads[,1]
heads[,1:2]
heads[,]
```

To find the number of columns in a data frame, use the function `length`. To find the number of rows, use `nrow`. We can also find the names of the columns by using `names`. Try these:

```
length(heads)
nrow(heads)
names(heads)
```

A.5 Making graphs

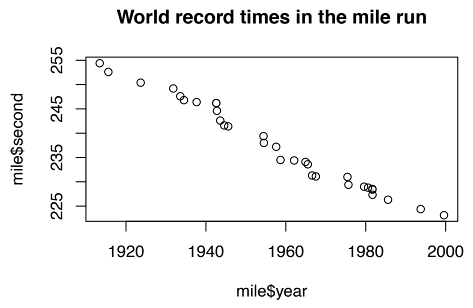
Graphing data

Example: Mile run

Scatterplots. A scatterplot shows the relation between two variables. The data frame called `mile` in the folder `Mile` contains the world record times in the mile run since 1900 as four columns, `yr`, `month`, `min`, and `sec`. For convenience we create the following derived quantities in R:

```
mile$year <- mile$yr + mile$month/12
mile$seconds <- mile$min*60 + mile$sec
```

Figure A.1 plots the world record time (in seconds) against time. We can create scatterplots in R using `plot`. Here is the code to make the basic graph:



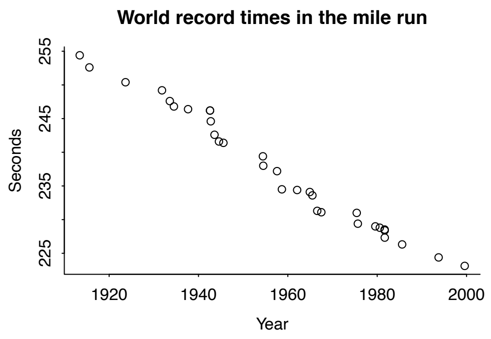


Figure A.1 *World record times (in minutes) in the mile run since 1900. The left plot was made with the basic function call:* plot(mile$year, mile$seconds, main="World record times in the mile run"). *The right plot was made with more formatting:* par(mar=c(3,3,3,1), mgp=c(2,.5,0), tck=-.01); plot(mile$year, mile$seconds, bty="l", main="World record times in the mile run", xlab="Year", ylab="Seconds").

```
plot(mile$year, mile$seconds, main="World record times in the mile run")
```

And here is a slightly prettier version:

```
par(mar=c(3,3,3,1), mgp=c(2,.5,0), tck=-.01)
plot(mile$year, mile$seconds, bty="l",
     main="World record times in the mile run", xlab="Year", ylab="Seconds")
```

The `par` function sets graphical parameters, in this case reducing the blank border around the graph, placing the labels closer to the axes, and reducing the size of the tick marks, compared to the default settings in R. In addition, in the call to `plot`, we have set the box type to `l`, which makes an "L-shaped" box for the graph rather than fully enclosing it.

We want to focus on the basics, so for the rest of this section we will show simple `plot` calls that won't make such pretty graphs. But we thought it would be helpful to show one example of a pretty graph, hence the code just shown above. The folder `Mile` also includes an example of making a pretty graph using the `ggplot2` package.

Fitting a line to data. We can fit a linear trend of world record times as follows:

```
fit <- stan_glm(seconds ~ year, data=mile)
print(fit)
```

Here is the result:

```
            Median MAD_SD
(Intercept) 1006.1   23.3
year          -0.4    0.0

Auxiliary parameter(s):
      Median MAD_SD
sigma 1.4    0.2
```

Interpretation of these results is discussed elsewhere in the book. All you need to know now is that the estimated coefficients are 1006.1 and −0.4; that is, the fitted regression line is $y = 1006.1 - 0.4 * \text{year}$. It will help to have more significant digits on this slope, so we type `print(fit, digits=2)` to get:

```
            Median   MAD_SD
(Intercept) 1006.15   23.33
year          -0.39    0.01

Auxiliary parameter(s):
      Median MAD_SD
sigma 1.42   0.19
```

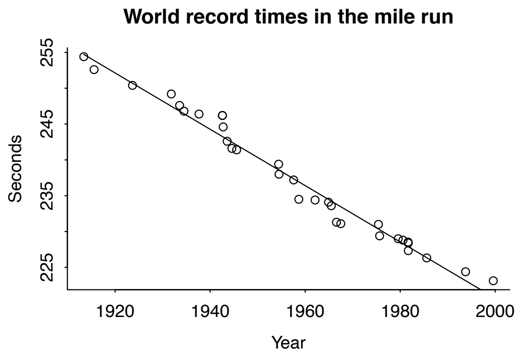


Figure A.2 *Fitted line predicting the world record time in the mile run given year, overlaying the data shown in Figure A.1b, plotted by adding the line,* curve(coef(fit)[1] + coef(fit)[2]*x, add=TRUE)*, to that code.*

The estimated line is $y = 1006.15 - 0.39 * \text{year}$.

We can add the straight line to the scatterplot by adding the following line after the call to `plot`:

```
curve(1006.14 - 0.39*x, add=TRUE)
```

The first argument is the equation of the line as a function of x. The second argument, `add=TRUE`, tells R to draw the line onto the existing scatterplot. More cleanly, we can extract the parameters for the curve from the fitted regression:

```
curve(coef(fit)[1] + coef(fit)[2]*x, add=TRUE)
```

The result is displayed in Figure A.2.

Multiple graphs on a page

Visualizations can be much more powerful using *small multiples*: repeated graphs on a similar theme. There are various ways to put multiple graphs on a page in R; one way uses the `par` function with its `mfrow` option, which tells R to lay out graphs in a grid, row by row.

We illustrate in Figure A.3 with a simple example plotting random numbers:

```
par(mfrow=c(2,4))
for (i in 1:2){
  for (j in 1:4){
    x <- rnorm(10)
    y <- rnorm(10)
    plot(x, y, main=paste("Row", i, "Column", j))
  }
}
```

The actual code we used to make the graphs also includes instructions to label the graphs, put them on a common scale, and size them to better fit on the page, but the above code gives the basic idea.

A.6 Working with messy data

Reading in survey data, one question at a time

Example: Height and weight

Data on the heights, weights, and earnings of a random sample of Americans are available from the Work, Family, and Well-Being Survey conducted by Catherine Ross in 1990. We downloaded the data file, `06666-0001-Data.txt`, and the codebook, `06666-0001-Codebook.txt`, from the Inter-university Consortium for Political and Social Research. Information on the survey is at `dx.doi.org/10.3886/ICPSR06666` and can be downloaded from the ICPSR.

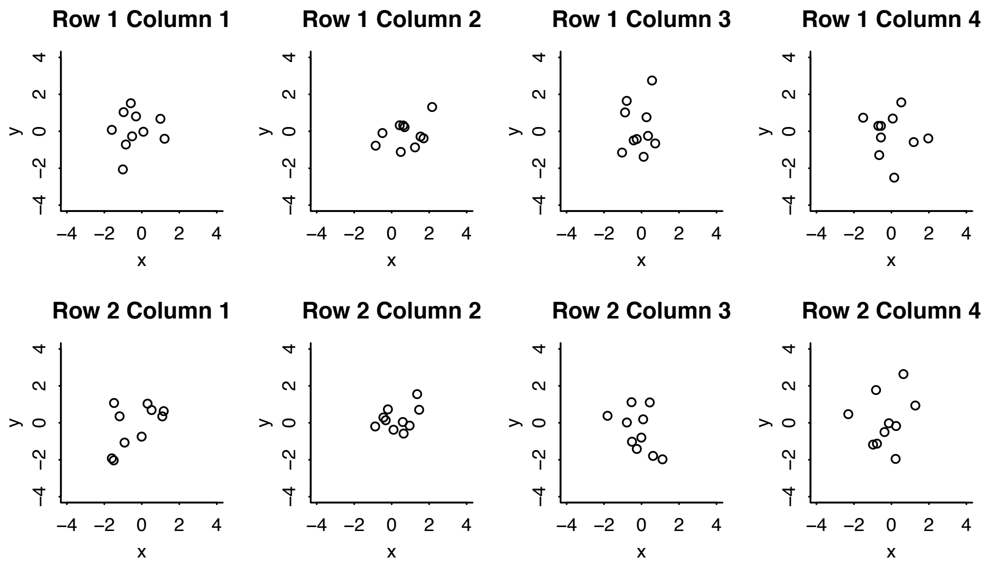


Figure A.3 *Example of a grid of graphs of random numbers produced in R. Each graph plots 10 pairs of numbers randomly sampled from the 2-dimensional normal distribution, and the display has been formatted to make 20 plots visible on a single page with common scales on the x and y-axes and customized plot titles.*

We saved the files under the names `wfw90.dat` and `wfwcodebook.txt` in the `Earnings` folder.

Figure A.4 shows the first 10 lines of the data, and Figure A.5 shows the relevant portion of the codebook. Our first step is to save the data file `wfwcodebook.txt` in our working directory. We then want to extract the responses to the questions of interest. To do this we first create a simple function to read columns of data, making use of R's function `read.fwf` (read in fixed width format).

Copy the following into the R console and then you will be able to use our function for reading one variable at a time from the survey. The following code is a bit tricky, so you're not expected to understand it; you can just copy it in and use it.

```
read.columns <- function (filename, columns) {
  start <- min(columns)
  length <- max(columns) - start + 1
  if (start==1)
    return(read.fwf(filename, widths=length))
  else
    return(read.fwf(filename, widths=c(start-1, length))[,2])
}
```

The data do not come in a convenient comma-separated or tab-separated format, so we use the function `read.columns` to read the coded responses, one question at a time:

```
height_feet <- read.columns("wfw90.dat", 144)
height_inches <- read.columns("wfw90.dat", 145:146)
weight <- read.columns("wfw90.dat", 147:149)
earn_exact <- read.columns("wfw90.dat", 203:208)
earn2 <- read.columns("wfw90.dat", 209:210)
sex <- read.columns("wfw90.dat", 219)
```

Cleaning data within R

We now must put the data together in a useful form, doing the following for each variable of interest:

1. Look at the data.

```
100022  31659123     121222113121432 22  2 3 411179797979797 1  4 503100100...
100081  486 2122     111222141122221222  2 1 997979797979797 1  4  01 25 25...
100091   1371123     123212211113111314 1 0                         30100...
100101  15684222     133122121113232 22  1 0                         10 40...
100111  25371122     122222111111421222  2 2 4 6979797979797 1  2 853 30 95...
100202   2389013     1111221412   22 314 2 0                        100100...
100281   7884021     2232132422   42 22  2 0                         80 60...
100351  15684221     233223242112212 32  2 0                         75100...
100571  88341221     243233321113452 12  2 0                         90 98...
100641  15684223     122112211113432 22  2 0                        100 75...
```

Figure A.4 *First 10 lines of the file* wfw90.dat*, which has data from the Work, Family, and Well-Being Survey.*

```
    HEIGHT     144-146   F3.0     Q.46 HEIGHT IN INCHES
    WEIGHT     147-149   F3.0     Q.47 WEIGHT
. . .
    EARN1      203-208   F6.0     Q.61  PERSONAL INCOME - EXACT AMOUNT
    EARN2      209-210   F2.0     Q.61  PERSONAL INCOME - APPROXIMATION
    SEX        219       F1.0     Q.63  GENDER OF RESPONDENT
. . .
    HEIGHT
    46. What is your height without shoes on?
       __________ ft.  __________in.
    WEIGHT
    47. What is your weight without clothing?
       __________ lbs.
. . .
    61a. During 1989, what was your personal income from your own wages,
    salary, or other sources, before taxes?
    EARN1
    $ __________--> (SKIP TO Q-62a)
    DON'T KNOW . . . 98
    REFUSED  . . . . 99
```

Figure A.5 *Selected rows of the file* wfwcodebook.txt*, which first identifies the columns in the data corresponding to each survey question and then gives the question wordings.*

2. Identify errors or missing data.
3. Transform or combine raw data into summaries of interest.

We start with `table(sex)`, which simply yields:

```
sex
   1    2
 749 1282
```

No problems. But we prefer to have a more descriptive name, so we define a new variable, `male`:

```
male <- 2 - sex
```

This *indicator variable* equals 0 for men and 1 for women.

We next look at height, typing: `table(height_feet, height_inches)`. Here is the result:

```
           height_inches
height_feet   0   1   2   3   4   5   6   7   8   9  10  11  98  99
          4   0   0   0   0   0   0   0   0   0   1   3  17   0   0
          5  66  56 144 173 250 155 247 127 174 105 145  90   0   0
          6 129  59  46  20   8   5   1   0   0   0   1   0   0   0
          7   0   0   0   0   0   0   0   1   0   0   0   0   0   0
          9   0   0   0   0   0   0   0   0   0   0   0   0   2   6
```

Most of the data look fine, but there are some people with 9 feet and 98 or 99 inches (missing data codes) and one person who is 7 feet 7 inches tall (probably a data error). We recode as missing:

```
height_inches[height_inches>11] <- NA
height_feet[height_feet>=7] <- NA
```

And then we define a combined height variable:

```
height <- 12*height_feet + height_inches
```

Next, we type `table(weight)` and get the following:

```
weight
 80  85  87  89  90  92  93  95  96  98  99 100 102 103 104 105 106 107 108 110
  1   1   1   1   1   1   1   2   2   3   1  12   5   4   3  16   1   5   7  46
111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130
  4  15   5   5  42   5   4  21   4  72   4  14  20  11  61  11   3  25   8 106
131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
  4  16   9   5  85   9  10  15   4  94   1  12   2   4  74   2   7   8   5 121
151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
  2   4   8   7  49   3   6  14   1  88   1   8   4   9  65   2   2   8   2  81
171 172 173 174 175 176 178 180 181 182 183 184 185 186 187 188 189 190 192 193
  2  10   4   2  58   5   4  78   3   4   2   4  62   1   5   1   3  46   1   3
194 195 196 197 198 199 200 201 202 203 205 206 207 208 209 210 211 212 214 215
  4  26   3   2   3   2  57   1   2   2  11   2   3   2   3  36   1   2   3  10
217 218 219 220 221 222 223 225 228 230 231 235 237 240 241 244 248 250 255 256
  1   1   1  21   2   1   1  13   2  17   1   3   1  13   1   1   1  10   3   1
260 265 268 270 275 280 295 312 342 998 999
  2   3   1   3   1   3   1   1   1   6  36
```

Everything looks fine until the end. The numbers 998 and 999 must be missing data, which we duly code as such:

```
weight[weight>500] <- NA
```

Coding the earnings responses is more complicated. The variable `earn_exact` contains responses (in dollars per year) for those who answered the question. Typing `table(is.na(earn_exact))` reveals that 1380 people answered the question (that is, `is.na(earn_exact)` is `FALSE` for these respondents), and 651 did not answer (`is.na` is `TRUE`). These nonrespondents were asked a second, discrete, earnings question, `earn2`, which gives earnings in round numbers (in thousands of dollars per year) for people who were willing to answer in this way. A careful look at the codebook reveals a set of nonreponse codes requring recoding of the data; details appear in the file `earnings_setup.R` in the `Earnings` folder. In particular, a code of 90 represents nonresponse or refusal to answer, and a code of 1 corresponds to people who did not supply an exact earnings value but did say it was more than \$100 000. For these people we perform a simple imputation using the median of the exact earnings responses that exceeded 100 000:

```
earn_approx[earn2>=90] <- NA
earn_approx[earn2==1] <- median(earn_exact[earn_exact>100000], na.rm=TRUE)/1000
```

We divided by 1000 because `earn_approx` is on the scale of thousands of dollars. We then create a combined earnings variable:

```
earn <- ifelse(is.na(earn_exact), 1000*earn_approx, earn_exact)
```

The new `earn` variable still has 211 missing values (out of 2031 respondents in total) and is imperfect in various ways, but we have to make some choices when working with real data.

Looking at the data

If you stare at the table of responses to the weight question you can see more. People typically round to the nearest 5 or 10 pounds, and so we see a lot of weights reported as 100, 105, 110, and so forth, but not so many in between. Beyond this, people appear to like round numbers: 57 people report weights of 200 pounds, compared to only 46 and 36 people reporting 190 and 210, respectively.

Similarly, if we go back to reported heights, we see some evidence that the reported numbers do not correspond exactly to physical heights: 129 people report heights of exactly 6 feet, compared to 90 people at 5 feet 11 inches and 59 people at 6 feet 1 inch. Who are these people? Let's look at the breakdown of height and sex:

```
table(male, height)
```

Here's the result:

```
     height
male  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75
   0   1   3  17  63  54 140 170 236 142 200  84  93  28  26   8   4   3   2   1
   1   0   0   0   3   2   4   3  14  13  47  43  81  77 119  82 125  56  44  19
    height
male  76  77  78  82
   0   0   0   0   0
   1   8   5   1   1
```

The extra 6-footers (72 inches tall) are just about all men. But there appear to be too many women of exactly 5 feet tall and an excess of both men and women who report being exactly 5 feet 6 inches.

A.7 Some R programming

Writing your own functions

You can write your own function. Most of the functions we will be writing will take in one or more vectors and return a vector. Below is an example of a simple function that triples the value provided:

```
triple <- function(x) {3*x}
```

For example, to call this function, type `triple(c(1,2,3,5))`. This function has one argument, `x`. The body of the function is within the curly braces and the arguments of the function are available for use within the braces. In our example function, we multiply `x` by 3 and we return it back to the user. If we wanted to have more than one argument, we could do something like this:

```
new_function <- function(x, y, a, b) {
  a*x+b*y
}
```

Optimization

Example: Simple optimization

Finding the peak of a parabola. Figure A.6 shows the parabola $y = 15 + 10x - 2x^2$. As you can see, the peak is at $x = 2.5$. How can we find this solution systematically? Finding the maximum of a function is called an *optimization* problem. Here's how we do it in R and Stan.

1. Plot the function, which you can do using `curve(15+10*x-2*x^2,from=___,to=___)`, entering numbers in the blanks. If necessary, you can play around with the "from" and "to" arguments until the maximum appears on the graph.
2. Write it as a Stan program called `parabola.stan`:[1]

[1]Code for this example is in the folder `Parabola`.

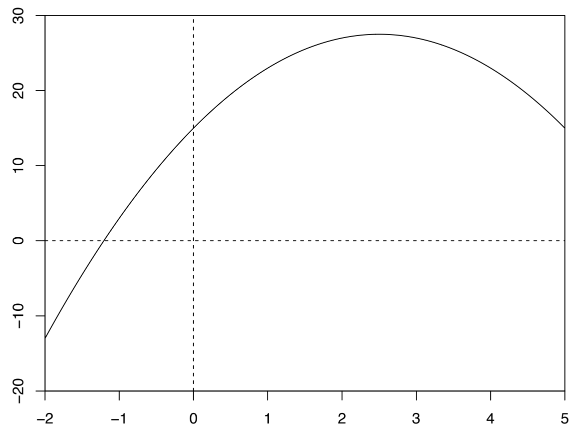


Figure A.6 *The parabola $y = 15 + 10x - 2x^2$, plotted in R using* curve(15 + 10*x - 2*x^2, from=-2, to=5). *The maximum is at $x = 2.5$.*

```
parameters {
  real x;
}
model {
  target += 15 + 10*x - 2*x^2;
}
```

3. In R, compile the Stan function and optimize it:

```
library("rstan")
model <- stan_model("parabola.stan")
fit <- optimizing(model)
print(fit)
```

This yields:

```
$par
  x
2.5

$value
[1] 27.5

$return_code
[1] 0
```

The output labeled `par` is the value of x at which the function is optimized, and the output labeled `value` is the target function at the optimum. A `return_code` of 0 corresponds to the optimizer running with no problem, and you can ignore `theta_tilde`.

4. Check the solution on the graph to see that it makes sense.

Example: Restaurant pricing

Restaurant pricing. Suppose you own a small restaurant and are trying to decide how much to charge for dinner. For simplicity, suppose that dinner will have a single price, and that your marginal cost per dinner is \$11. From a marketing survey, you have estimated that, if you charge \$$x$ for dinner, the average number of customers per night you will get is $5000/x^2$. How much should you charge, if your goal is to maximize expected net profit?

Your expected net profit is the expected number of customers times net profit per customer, that is, $(5000/x^2) * (x - 11)$. To get a sense of where this is maximized, we can first make a graph:

```
net_profit <- function(x){
  (5000/x^2)*(x-11)
}
curve(net_profit(x), from=10, to=100, xlab="Price of dinner",
  ylab="Net profit per night")
```

From a visual inspection of this curve, the peak appears to be around $x = 20$. There are two ways we can more precisely determine where net profit is maximized.

First, the brute-force approach. The optimum is clearly somewhere between 10 and 100, so let's just compute the net profit at a grid of points in this range:

```
x <- seq(10, 100, 0.1)
y <- net_profit(x)
```

The maximum value of net profit is then simply `max(y)`, which equals 113.6, and we can find the value of x where profit is maximized:

```
x[y==max(y)]
```

It turns out to be 22, which is `x[121]`, the 121st element of x. Above, we are subscripting the vector `x` with a logical vector `y==max(y)`, which is a vector of 900 `FALSE`'s and one `TRUE` at the maximum.

We can also directly optimize using Stan. First we write the function as a Stan program and save it in the file `restaurant.stan`:[2]

```
parameters {
  real<lower=0,upper=100> x;
}
model {
  target += (5000/x^2)*(x-11);
}
```

Then, in R:

```
resto <- stan_model("restaurant.stan")
fit <- optimizing(resto)
print(fit)
```

The result:

```
$par
      x
22.0

$value
[1] 113.6

$return_code
[1] 0
```

The function returns 113.6 at the input value $x = 22.0$. If the return code had not been zero, that would have indicated a problem with the optimization.

A.8 Working with rstanarm fit objects

Cleaner output

By default, `stan` and `stan_glm` print intermediate output into the R console. Including the argument `refresh=0` suppresses this output.

[2]Code for this example is in the folder `Restaurant`.

Extracting posterior summaries, simulation draws, and predictions

We have collected here some useful functions for obtaining information or manipulating fitted models from `stan_glm`—these are called `stanreg` objects. As an example we use the children's test score model introduced in Chapter 10.

```
fit <- stan_glm(kid_score ~ mom_hs + mom_iq, data=kidiq)
```

Most simply, `print(fit)` displays the parameter estimates and associated uncertainties:

```
            Median MAD_SD
(Intercept) 25.7    5.9
mom_hs       6.0    2.4
mom_iq       0.6    0.1

Auxiliary parameter(s):
      Median MAD_SD
sigma 18.2    0.6
```

The coefficients are always displayed first, with auxiliary parameters at the end. In linear regression, the only auxiliary parameter is the residual standard deviation, σ. Logistic regression has no auxiliary parameters at all. Other generalized linear models can have various auxiliary parameters such as overdispersions and cutpoints.

Markov chain Monte Carlo diagnostics are provided by `summary(fit)`:

```
Diagnostics:
              mcse Rhat n_eff
(Intercept)   0.1  1.0  5021
mom_hs        0.0  1.0  4663
mom_iq        0.0  1.0  4557
sigma         0.0  1.0  5115
mean_PPD      0.0  1.0  4443
log-posterior 0.0  1.0  1926
```

Bayesian R^2 (see Section 11.6) can be extracted using `bayes_R2(fit)`, which returns posterior draws of Bayesian R^2. A point summary can be obtained, for example, using `median(bayes_R2(fit))`.

To check the priors used for the inference, use `prior_summary(fit)`. Default priors can be replaced when calling `stan_glm`. For more information, type `vignette("priors", package="rstanarm")` in the R console.

When fitted models are used within programs, it can be helpful to extract the estimated vector of coefficients and residual standard deviation using `coef(fit)` and `sigma(fit)`.

You can extract the estimated residual standard deviation using `se(fit)`, which can be helpful when quickly plotting inferences (as in Figure 19.4); just remember that the estimates and standard errors do *not* fully capture inferential uncertainty, as they do not encode posterior correlation of the parameters. When propagating uncertainty in coefficients, it is best to extract the posterior simulations directly, which can be done using `as.matrix(fit)` or `as.data.frame(fit)`.

As discussed in Sections 9.2 and 13.3, predictions can be obtained with different levels of uncertainty: `predict(fit)` returns point predictions, and `posterior_linpred(fit)`, `posterior_epred(fit)`, and `posterior_predict(fit)` return draws from the posterior distribution of the linear predictor, the expected value of data, and the posterior predictive distribution, respectively. If you call these predictive functions without supplying a `newdata` argument, they are automatically applied to the existing data, so that `predict` returns a vector of n point predictions, while `posterior_linpred(fit)`, `posterior_epred(fit)`, and `posterior_predict(fit)` each return a $n_{\text{sims}} \times n$ matrix of simulations. If you supply a `newdata` argument that is a data frame with $\tilde{n}$ new observations—for that data frame you just need to supply the predictors for these new data—then `predict` returns a vector of length $\tilde{n}$, and `posterior_linpred(fit)`, `posterior_epred(fit)`, and `posterior_predict(fit)` each return a $n_{\text{sims}} \times \tilde{n}$ matrix.

To find more functions that accept a `stanreg` object, type `methods(class=class(fit))` from the R console.

R packages for visualizing and post-processing stanreg objects

The `shinystan` package (`mc-stan.org/shinystan`) provides a graphical user interface for interactive Markov chain Monte Carlo (MCMC) diagnostics and plots and tables helpful for analyzing a posterior sample. Simply run `launch_shinystan(fit)` after model fitting.

The `bayesplot` package (`mc-stan.org/bayesplot`) provides plotting functions for posterior analysis, model checking, and MCMC diagnostics using `ggplot2` graphics. In the examples in the book, we have used the functions `mcmc_areas` and `mcmc_scatter` to make marginal posterior, and bivariate marginal posterior plots. In Chapter 11 we used `ppc_hist`, `ppc_dens_overlay`, and `ppc_stat` to make posterior predictive checking plots. The `posterior` and `tidybayes` packages provide further functions for posterior analysis, data manipulation, and graphics for Bayesian inference.

Fitting Stan models in R using brms

The `brms` package (`github.com/paul-buerkner/brms`) uses an extended version of the formula syntax used by `rstanarm`. Unlike `rstanarm`, `brms` does not have pre-compiled models, which allows more flexibility in the formula syntax, but on the other hand requires C++ compilation, making interactive use slower. The models used in this book also work with `brms`, just by replacing `stan_glm` with `brm`. The default prior distributions for `brms` are slightly different, but this does not cause major differences for our examples. Using the function `make_stancode`, the `brms` package can be used to produce Stan model code, which can make it easier to learn the Stan language and start writing new models which can then be modified by editing that code.[3]

A.9 Bibliographic note

The books by Becker, Chambers, and Wilks (1988); Venables and Ripley (2002); Wickham (2014); and Wickham and Grolemund (2017) provide thoughtful overviews of R, and its predecessor S, from different perspectives. R is currently maintained by R Core Team (2019).

Stan is introduced by Gelman, Lee, and Guo (2015) from a users' perspective and Carpenter et al. (2017) from a developers' perspective. Much more documentation on R and Stan is available at `www.r-project.org` and `www.mc-stan.org`. Stan is currently maintained by Stan Development Team (2020).

The following additonal free software has been used in the examples: `rstanarm` (Goodrich et al., 2019), `brms` (Buerkner, 2017, 2018), `bayesplot` (Gabry and Mahr, 2019), `shinystan` (Gabry, 2018), `loo` (Vehtari et al., 2019), `arm` (Gelman and Su, 2018), `MASS` (Venables and Ripley, 2002), `survey` (Lumley, 2019), `BART` (McCulloch et al., 2019), `ggplot2` (Wickham, 2016), `knitr` (Xie, 2015, 2019), and `rprojroot` (Mueller, 2018); see also `posterior` (Buerkner et al., 2020) and `tidybayes` (Kay, 2020).

The Work, Family, and Well-Being Survey is described by Ross (1990).

[3]Code to illustrate this is in the folder `DifferentSoftware`.