

# R Programming for Data Sciences

Vince Melfi & Andrew O. Finley

August 26, 2018

# Contents

<b>1</b>	<b>Data</b>	<b>1</b>
1.1	Baby crawling data . . . . .	1
1.2	World Bank Data . . . . .	1
1.3	Email data . . . . .	3
1.4	Handwritten digit recognition . . . . .	5
1.5	Looking forward . . . . .	9
1.6	How to learn (The most important section in this book!) . . . . .	9
<b>2</b>	<b>Introduction to R and RStudio</b>	<b>11</b>
2.1	Obtaining and installing R . . . . .	11
2.2	Obtaining and installing RStudio . . . . .	12
2.3	RStudio Server . . . . .	12
2.4	Using R and RStudio . . . . .	14
2.4.1	R as a calculator . . . . .	15
2.4.2	Basic descriptive statistics and graphics in R . . . . .	16
2.4.3	An initial tour of RStudio . . . . .	19
2.5	How to learn (The most important section in this book!) . . . . .	20
2.6	Getting help . . . . .	20
2.7	Workspace, working directory, and keeping organized . . . . .	21
2.8	Quality of R code . . . . .	22
<b>3</b>	<b>Scripts, R Markdown, and reproducible research</b>	<b>24</b>
3.1	Scripts in R . . . . .	25
3.2	R Markdown . . . . .	31
3.2.1	Creating and processing R Markdown documents . . . . .	33
Text: Lists and Headers . . . . .	34	
Code Chunks . . . . .	36	
3.2.2	Output formats other than HTML . . . . .	40

<b>CONTENTS</b>	<b>2</b>
3.2.3 L <sup>A</sup> T <sub>E</sub> X and knitr . . . . .	40
<b>4 Data Structures</b>	<b>41</b>
4.1 Vectors . . . . .	42
4.1.1 Types, conversion, coercion . . . . .	43
Coercion . . . . .	45
4.1.2 Accessing specific elements of vectors . . . . .	46
4.2 Factors . . . . .	48
4.3 Names of objects in R . . . . .	49
4.4 Missing Data, Infinity, etc. . . . .	50
4.4.1 Infinity and NaN . . . . .	51
4.5 Data Frames . . . . .	52
4.5.1 Accessing specific elements of data frames . . . . .	54
4.6 Lists . . . . .	55
4.6.1 Accessing specific elements of lists . . . . .	58
4.7 Subsetting with logical vectors . . . . .	60
4.7.1 Modifying or creating objects via subsetting . . . . .	61
4.7.2 Logical subsetting and data frames . . . . .	62
4.8 Patterned data . . . . .	67
<b>5 Graphics in R</b>	<b>70</b>
5.1 Scatter Plots . . . . .	71
5.1.1 Structure of a typical ggplot . . . . .	74
5.1.2 Adding lines to a scatter plot . . . . .	76
5.2 Labels, axes, text, etc. . . . .	82
5.2.1 Labels . . . . .	84
5.2.2 Customizing axes . . . . .	85
5.2.3 Text, point size, and color . . . . .	87
5.3 Other types of graphics . . . . .	90
5.3.1 Histograms . . . . .	90
5.3.2 Boxplots . . . . .	92
5.3.3 Bar graphs . . . . .	94
5.3.4 Graphs of functions . . . . .	99
5.4 Themes . . . . .	100
5.5 Saving graphics . . . . .	101
5.6 More resources . . . . .	102

<b>6 Working with Data</b>	<b>104</b>
6.1 Reading data into R . . . . .	104
6.1.1 Reading data with missing observations . . . . .	107
6.2 Summarizing data frames . . . . .	109
6.2.1 Column (and row) summaries . . . . .	109
6.2.2 The <code>apply()</code> function . . . . .	112
6.2.3 Saving typing using <code>with()</code> . . . . .	113
6.3 Transforming a data frame . . . . .	113
6.3.1 Adding variables . . . . .	114
6.3.2 Removing variables . . . . .	115
6.3.3 Transforming variables . . . . .	117
6.3.4 Rearranging variables . . . . .	117
6.4 Reshaping data . . . . .	119
6.4.1 <code>Tidyr</code> . . . . .	119
6.5 Manipulating data with <code>dplyr</code> . . . . .	123
6.5.1 Improved data frames . . . . .	124
6.5.2 Filtering data by row . . . . .	126
6.5.3 Selecting variables by column . . . . .	129
6.5.4 Pipes . . . . .	131
6.5.5 Arranging data by row . . . . .	132
6.5.6 Renaming variables . . . . .	134
6.5.7 Data summaries and grouping . . . . .	135
6.5.8 Creating new variables . . . . .	140
<b>7 Functions and Programming</b>	<b>144</b>
7.1 R functions . . . . .	144
7.1.1 Creating functions . . . . .	148
7.2 Programming: Conditional Statements . . . . .	148
7.2.1 Comparison and logical operators . . . . .	150
7.2.2 If else statements . . . . .	151
7.3 Computer Arithmetic . . . . .	154
7.4 Loops . . . . .	155
7.4.1 A repeat loop . . . . .	156
7.4.2 A while loop . . . . .	157
7.4.3 A for loop . . . . .	157
7.5 Efficiency considerations . . . . .	158
7.5.1 Growing objects . . . . .	158

7.5.2	Vectorization . . . . .	159
7.6	More on functions . . . . .	161
7.6.1	Calling functions . . . . .	161
7.6.2	The ... argument . . . . .	162
7.6.3	Lazy evaluation . . . . .	164
<b>8</b>	<b>Simulation</b>	<b>166</b>
8.1	Pseudo-random numbers . . . . .	166
8.2	Simulating from standard distributions . . . . .	171
8.3	Investigating <i>t</i> tests . . . . .	173
8.4	Exercise . . . . .	183
<b>9</b>	<b>Classification</b>	<b>185</b>
9.1	Logistic regression . . . . .	185
9.1.1	Adding predictors . . . . .	193
9.1.2	More than two classes . . . . .	196
9.2	Nearest neighbor methods . . . . .	201
9.2.1	kNN and the diabetes data . . . . .	208
9.2.2	kNN and the iris data . . . . .	209
<b>10</b>	<b>Text Data</b>	<b>211</b>
10.1	Reading text data into R . . . . .	211
10.2	The <code>paste</code> function . . . . .	216
10.3	More string processing functions . . . . .	221
<b>11</b>	<b>Spatial data visualization and analysis</b>	<b>231</b>
11.1	Overview . . . . .	231
11.1.1	Some spatial data packages . . . . .	232
11.2	Motivating data . . . . .	233
11.3	Reading spatial data into R . . . . .	233
11.4	Coordinate reference systems . . . . .	238
11.5	Illustration using <code>ggmap</code> . . . . .	239
11.6	Illustration using <code>leaflet</code> . . . . .	248
11.7	Subsetting spatial data . . . . .	249
11.7.1	Fetching and cropping data using <code>raster</code> . . . . .	251
11.7.2	Logical, index, and name subsetting . . . . .	254
11.7.3	Spatial subsetting and overlay . . . . .	255
11.7.4	Spatial aggregation . . . . .	259

11.8 Where to go from here . . . . .	262
<b>12 Shiny</b>	<b>264</b>
12.1 Running a Simple Shiny App . . . . .	264
12.2 Adding User Input . . . . .	267
12.3 Adding Output . . . . .	268
12.3.1 Interactive Server Logic . . . . .	268
12.3.2 Interactive User Interface . . . . .	269
12.4 More Complicated Shiny App: Michigan Campgrounds . . . . .	270
12.4.1 Michigan Campgrounds UI . . . . .	271
12.4.2 Michigan Campgrounds Server Logic . . . . .	271
12.5 Adding Leaflet to Shiny . . . . .	272
12.6 Why use Shiny? . . . . .	273
<b>13 Rcpp</b>	<b>275</b>
13.1 Getting Started with Rcpp . . . . .	275
13.1.1 Installation . . . . .	275
13.1.2 The Simplest C++ Example . . . . .	276
13.2 Using Rcpp . . . . .	276
13.2.1 Exporting C++ Functions . . . . .	276
13.2.2 Inline C++ Code . . . . .	278
13.3 The Rcpp Interface . . . . .	278
13.3.1 No input and scalar output . . . . .	279
13.3.2 Scalar input and scalar output . . . . .	280
13.3.3 Vector input and scalar output . . . . .	280
13.3.4 Vector input and vector output . . . . .	282
13.3.5 Matrix input and vector output . . . . .	283
13.3.6 Matrix input and matrix output . . . . .	284

# List of Figures

1.1	A spam email message . . . . .	3
1.2	A non-spam email message . . . . .	4
1.3	A digitized version of a handwritten “6” . . . . .	6
1.4	The first 25 handwritten numerals, digitized . . . . .	7
1.5	The first 25 numeral sevens, digitized . . . . .	8
2.1	The RStudio IDE. . . . .	14
2.2	xkcd: Code Quality . . . . .	22
3.1	A script window in RStudio. . . . .	31
3.2	R Markdown file which produces the output in Figure 3.3. . . . .	34
3.3	Output from the R Markdown file in Figure 3.2. . . . .	34
3.4	R Markdown file to produce unordered and ordered lists. . . . .	35
3.5	Output from the R Markdown file in Figure 3.4. . . . .	35
3.6	R Markdown which produces the output in Figure 3.7. . . . .	36
3.7	Output from the R Markdown file in Figure 3.6. . . . .	36
3.8	Other useful L <sup>A</sup> T <sub>E</sub> X expressions and symbols available for use in R Markdown. . . . .	37
3.9	R Markdown illustrating some of the options for code chunks. . . . .	38
3.10	Output from the R Markdown in Figure 3.9. . . . .	39
10.1	A spam email message . . . . .	212
10.2	The beginning of a plain text file from Project Gutenberg containing the novel <i>Moby Dick</i> . . . . .	213
11.1	Raster/vector comparison (Wegmann [2010]) . . . . .	232
11.2	leaflet interactive web map. . . . .	250
12.1	running a Shiny app locally. . . . .	266
12.2	Your first Shiny app. . . . .	267

# List of Tables

1.1	Data on age at crawling . . . . .	2
1.2	A small portion of the World Bank data set . . . . .	2
4.1	Dimension and type content of base data structures in R. . . . .	41
11.1	An abbreviated list of <code>sp</code> and <code>raster</code> data objects and associated classes for the fundamental spatial data types. . . . .	234
12.1	Interactive elements . . . . .	271

# List of Exercises

1	Learning objectives: practice setting up a working directory and read in data; explore the workspace within RStudio and associated commands; produce basic descriptive statistics and graphics. . . . .	40
2	Learning objectives: practice working within RStudio; create a R Markdown document and resulting html document in RStudio; calculate descriptive statistics and graphics. . . . .	40
3	Learning objectives: create, subset, and manipulate vector contents and attributes; summarize vector data using R <code>table</code> and other functions; generate basic graphics using vector data. . . . .	68
4	Learning objectives: use functions to describe data frame characteristics; summarize and generate basic graphics for variables held in data frames; apply the subset function with logical operators; illustrate <code>NA</code> , <code>NaN</code> , <code>Inf</code> , and other special values occur; recognize the implications of using floating point arithmetic with logical operators. . . . .	68
5	Learning objectives: practice with lists, data frames, and associated functions; summarize variables held in lists and data frames; work with R's linear regression <code>lm</code> function output; review logical subsetting of vectors for partitioning and assigning of new values; generate and visualize data from mathematical functions. . . . .	69
6	Learning objectives: practice using <code>ggplot2</code> functions; summarize variables using graphics; introduce <code>ggplot2</code> facets. . . . .	103
7	Learning objectives: introduce <code>with</code> , <code>tapply</code> , and <code>cut</code> functions; summarize data using the <code>table</code> function with logical subsetting; practice using factor data types. . . . .	143
8	Learning objectives: work with messy data; import data from an external spreadsheet; practice using functions in <code>tidyverse</code> and <code>ggplot2</code> . . . . .	143
9	Learning objectives: work with several key <code>dplyr</code> functions; manipulate data frames (actually tibbles); summarize and visualize data from large data files. . . . .	143

10	Learning objectives: translate statistical notation into coded functions; learn about tools for checking validity of function arguments; practice writing functions that return multiple objects. . . . .	165
11	Learning objectives: explore the logistic regression classification method; apply the kNN classification method; create confusion matrices to compare classification methods; plot classified data. . . . .	184
12	Learning objectives: explore the logistic regression classification method; apply the kNN classification method; create confusion matrices to compare classification methods; plot classified data. . . . .	210
13	Learning objectives: read and write text data; concatenate text with the <code>paste</code> function; analyze text with <code>nchar</code> ; practice with functions; manipulate strings with <code>substr</code> and <code>strtrim</code> . . . . .	230
14	Learning objectives: practice loading and reprojecting spatial data; analyze spatial data; create leaflet maps to convey analysis results; interpret analysis results. . . . .	263
15	Learning objectives: practice updating <code>ggplot2</code> plot aesthetics; modify Shiny HTML output; add an interactive Shiny element. . . . .	274
16	Learning objectives: practice using <code>Rcpp</code> to run a C++ function through R; use <code>microbenchmark()</code> to compare function performance. . . . .	284

# List of Videos

- |   |  |    |
|---|--|----|
| 1 | This video reviews chapter topics and provides additional details on using the RStudio and RStudio Server IDE. | 23 |
|---|--|----|

# Chapter 1

## Data

Data science is a field that intersects with statistics, mathematics, computer science, and a wide range of applied fields such as marketing, biology, and physics. As such it is hard to formally define data science, but obviously data is central to data science, and it is useful at the start to consider some types of data that are of interest.

### 1.1 Baby crawling data

When thinking about data, we might initially have in mind a modest-sized and uncomplicated data set, most of which is made up of numbers. As an example of such a data set, a study was done to assess the possible relationship between the age at which babies first begin to crawl and the temperature at the time of first crawling. Participants in the study were volunteers.<sup>1</sup> The data set from this study separates the babies by birth month, and reports the birth month, the average age (in weeks) when first crawling for that month, the standard deviation of the crawling ages for that month, the number of infants for that month, and the average temperature during the month when crawling commenced. The data are shown in Table 1.1 below.<sup>2</sup>

The size of this data set is relatively small, there are no missing observations, the variables are easily understood, etc.

### 1.2 World Bank Data

The World Bank provides data related to development of countries. A data set was constructed from the World Bank repository. The data set contains data on countries of the world for the years 1960 through 2014 and contains, among others, variables representing average life expectancy, fertility rate, and population. Table 1.2 contains the first five records and then 10 more randomly selected records for these variables in

---

<sup>1</sup>More correctly, were volunteered by their parents.

<sup>2</sup>These data were retrieved from <http://lib.stat.cmu.edu/DASL/Datafiles/Crawling.html>.

BirthMonth	AvgCrawlingAge	SD	n	temperature
January	29.84	7.08	32	66
February	30.52	6.96	36	73
March	29.70	8.33	23	72
April	31.84	6.21	26	63
May	28.58	8.07	27	52
June	31.44	8.10	29	39
July	33.64	6.91	21	33
August	32.82	7.61	45	30
September	33.83	6.93	38	33
October	33.35	7.29	44	37
November	33.38	7.42	49	48
December	32.32	5.71	44	57

Table 1.1: Data on age at crawling

the data set.

country	year	fertility.rate	life.expectancy	population
Andorra	1978			33746.00
Andorra	1979			34819.00
Andorra	1977			32769.00
Andorra	2007	1.18		81292.00
Andorra	1976			31781.00
Bulgaria	1969	2.27	70.43	8434172.00
St. Martin (French part)	1977			6778.00
Macao SAR, China	2007	0.91	79.06	493206.00
Iran, Islamic Rep.	1979	6.42	55.27	37465764.00
Comoros	1997	5.27	57.18	489627.00
Syrian Arab Republic	1975	7.47	62.78	7564000.00
Virgin Islands (U.S.)	1967	5.56	66.46	50800.00
Malta	1982	1.90	73.40	325898.00
St. Lucia	2013	1.91	74.79	182273.00
Vanuatu	1976	5.86	56.09	103024.00

Table 1.2: A small portion of the World Bank data set

Notice that many observations contain missing data for fertility rate and life expectancy. If all the variables were shown, we would see much more missing data. Also this data set is substantially larger than the baby crawling age data, with 11880 rows and 15 columns of data in the full data set. (Each column represents one of the variables. Each row represents one country during one year.)

```

From safety33o@111.newnamedns.com  Fri Aug 23 11:03:37 2002
Return-Path: <safety33o@111.newnamedns.com>
Delivered-To: zzzz@localhost.example.com
Received: from localhost (localhost [127.0.0.1])
    by phobos.labs.example.com (Postfix) with ESMTP id 5AC994415F
    for <zzzz@localhost>; Fri, 23 Aug 2002 06:02:59 -0400 (EDT)
Received: from mail.webnote.net [193.120.211.219]
    by localhost with POP3 (fetchmail-5.9.0)
    for zzzz@localhost (single-drop); Fri, 23 Aug 2002 11:02:59 +0100 (IST)
Received: from 111.newnamedns.com ([64.25.38.81])
    by webnote.net (8.9.3/8.9.3) with ESMTP id KAA09379
    for <zzzz@example.com>; Fri, 23 Aug 2002 10:18:03 +0100
From: safety33o@111.newnamedns.com
Date: Fri, 23 Aug 2002 02:16:25 -0400
Message-Id: <200208230616.g7N6GOR28438@111.newnamedns.com>
To: kxzzzzgxlrah@111.newnamedns.com
Reply-To: safety33o@111.newnamedns.com
Subject: ADV: Lowest life insurance rates available!
moodie

Lowest rates available for term life insurance! Take a moment
and fill out our online form
to see the low rate you qualify for.
Save up to 70% from regular rates! Smokers accepted!
http://www.newnamedns.com/termlife/

Representing quality nationwide carriers. Act now!

```

Figure 1.1: A spam email message

### 1.3 Email data

It is estimated that in 2015, 90% of the 205 billion emails sent each were spam.<sup>3</sup> Spam filters use large amounts of data from email to learn what distinguishes spam messages from non-spam (sometimes called “ham”) messages, Figures 1.1 and 1.2 contain one spam and one ham message, respectively.<sup>4</sup>

To implement a spam filter we would have to get the data from these (and thousands of other) email messages into a software package, extract and separate potentially important features such as the `To:` line, the `Subject:` line, the message body, etc., then compare spam and non-spam messages to find a method to classify new email correctly. None of these steps is simple in this example, and in particular we would need to become skilled at working with *text data*.

---

<sup>3</sup>Radicati Group [www.radicati.com](http://www.radicati.com).

<sup>4</sup>These messages both come from the large collection of spam and ham messages at <http://spamassassin.apache.org/publiccorpus/>.

From rssfeeds@jmason.org Tue Oct 1 10:37:22 2002  
Return-Path: <rssfeeds@example.com>  
Delivered-To: yyyy@localhost.example.com  
Received: from localhost (jalapeno [127.0.0.1])  
by jmason.org (Postfix) with ESMTP id B277816F16  
for <jm@localhost>; Tue, 1 Oct 2002 10:37:21 +0100 (IST)  
Received: from jalapeno [127.0.0.1]  
by localhost with IMAP (fetchmail-5.9.0)  
for jm@localhost (single-drop); Tue, 01 Oct 2002 10:37:21 +0100 (IST)  
Received: from dogma.slashnull.org (localhost [127.0.0.1]) by  
dogma.slashnull.org (8.11.6/8.11.6) with ESMTP id g9180YK15357 for  
<jm@jmason.org>; Tue, 1 Oct 2002 09:00:34 +0100  
Message-Id: <200210010800.g9180YK15357@dogma.slashnull.org>  
To: yyyy@example.com  
From: boingboing <rssfeeds@example.com>  
Subject: Disney's no-good Park-Czar replaced  
Date: Tue, 01 Oct 2002 08:00:34 -0000  
Content-Type: text/plain; encoding=utf-8  
X-Spam-Status: No, hits=-641.2 required=5.0  
tests=AWL  
version=2.50-cvs  
X-Spam-Level:  
  
URL: http://boingboing.net/#85506723  
Date: Not supplied

Disney has named a new president of Walt Disney Parks, replacing Paul Pressler, the exec who did his damnedest to ruin Disneyland, slashing spending (at the expense of safety and employee satisfaction), building the craptastical California Adventure, reducing the number of SKUs available for sale in the Park stores, and so on. The new president, James Rasulo, used to be head of Euro Disney. Link[1] Discuss[2]

- [1] [http://reuters.com/news\\_article.jhtml?type=search&StoryID=1510778](http://reuters.com/news_article.jhtml?type=search&StoryID=1510778)
- [2] <http://www.quicktopic.com/boing/H/rw7cDXT3W44C>

Figure 1.2: A non-spam email message

## 1.4 Handwritten digit recognition

Correct recognition of handwritten digits by a machine is commonly required. For example, the postal service must scan and recognize zip codes on handwritten mail. Roughly speaking, a handwritten digit is scanned and converted to a digital image. To keep things simple we will assume the scanning creates a grayscale rather than a color image. Typically converting to a digital image means overlaying a grid of “pixels” over the image and recording, for each pixel, the black intensity. For concreteness we’ll assume that intensities are recorded on a scale from  $-1$  (no black intensity at all) to  $1$  (maximum black intensity). If the pixel grid is  $16$  by  $16$  then the resulting digitized image will contain  $256$  intensity values, one for each of the  $16 \times 16 = 256$  pixels.

For example, here are the data corresponding to one handwritten digit, which happens to be the numeral “6.” Figure 1.3 shows how that digit looks when digitized.

```

-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.631  0.862 -0.167
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -0.992  0.297  1.000  0.307 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.410  1.000
  0.986 -0.565 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -0.683  0.825  1.000  0.562 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.938  0.540
  1.000  0.778 -0.715 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000  0.100  1.000  0.922 -0.439 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.257
  0.950  1.000 -0.162 -1.000 -1.000 -1.000 -0.987 -0.714 -0.832 -1.000
-1.000 -1.000 -1.000 -1.000 -0.797  0.909  1.000  0.300 -0.961 -1.000
-1.000 -0.550  0.485  0.996  0.867  0.092 -1.000 -1.000 -1.000 -1.000
  0.278  1.000  0.877 -0.824 -1.000 -0.905  0.145  0.977  1.000  1.000
  1.000  0.990 -0.745 -1.000 -1.000 -0.950  0.847  1.000  0.327 -1.000
-1.000  0.355  1.000  0.655 -0.109 -0.185  1.000  0.988 -0.723 -1.000
-1.000 -0.630  1.000  1.000  0.068 -0.925  0.113  0.960  0.308 -0.884
-1.000 -0.075  1.000  0.641 -0.995 -1.000 -1.000 -0.677  1.000  1.000
  0.753  0.341  1.000  0.707 -0.942 -1.000 -1.000  0.545  1.000  0.027
-1.000 -1.000 -1.000 -0.903  0.792  1.000  1.000  1.000  1.000  0.536
  0.184  0.812  0.837  0.978  0.864 -0.630 -1.000 -1.000 -1.000 -1.000
-0.452  0.828  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000
  0.135 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -0.483  0.813  1.000
  1.000  1.000  1.000  1.000  0.219 -0.943 -1.000 -1.000 -1.000 -1.000
-1.000 -1.000 -1.000 -1.000 -0.974 -0.429  0.304  0.823  1.000  0.482
-0.474 -0.991 -1.000 -1.000 -1.000 -1.000 -1.000

```

Looking at the digitized images, it may seem simple to correctly identify a handwritten numeral. But remember, the machine only has access to the  $256$  pixel intensities,

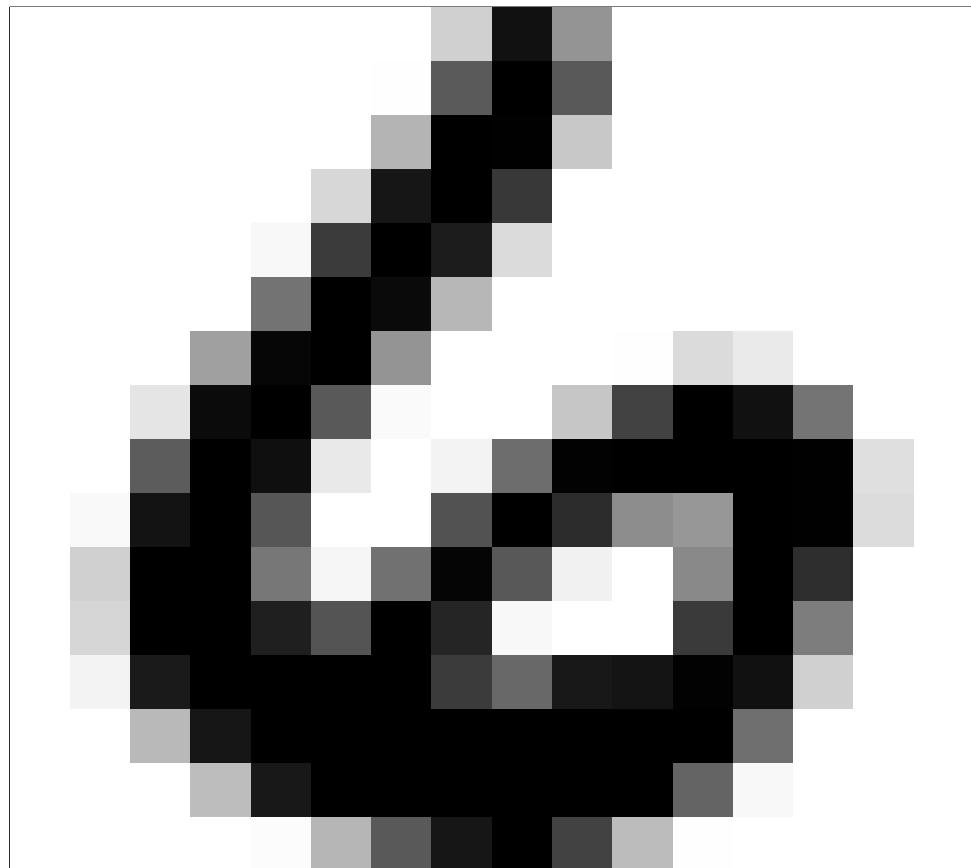


Figure 1.3: A digitized version of a handwritten “6”

and must make a decision based on them.

Figure 1.4 shows the digitized images of the first 25 numerals in the data set, and Figure 1.5 shows the digitized images of the first 25 numeral sevens in the data set. These give some idea of the variability in how digits are written.<sup>5</sup>

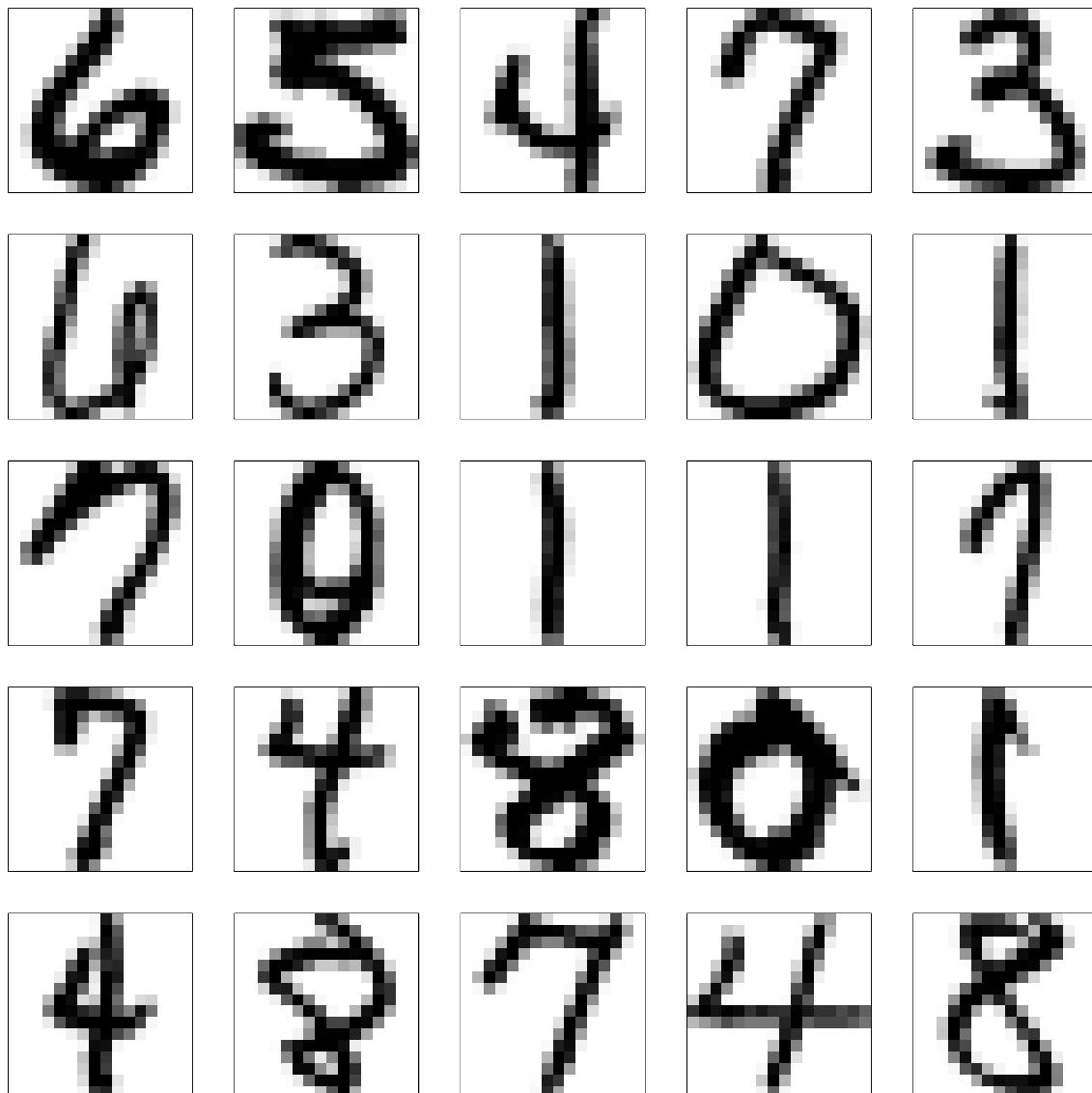


Figure 1.4: The first 25 handwritten numerals, digitized

---

<sup>5</sup> Actually, these data were already pre-processed to get the orientation correct. Actual handwritten digits would be even more variable.

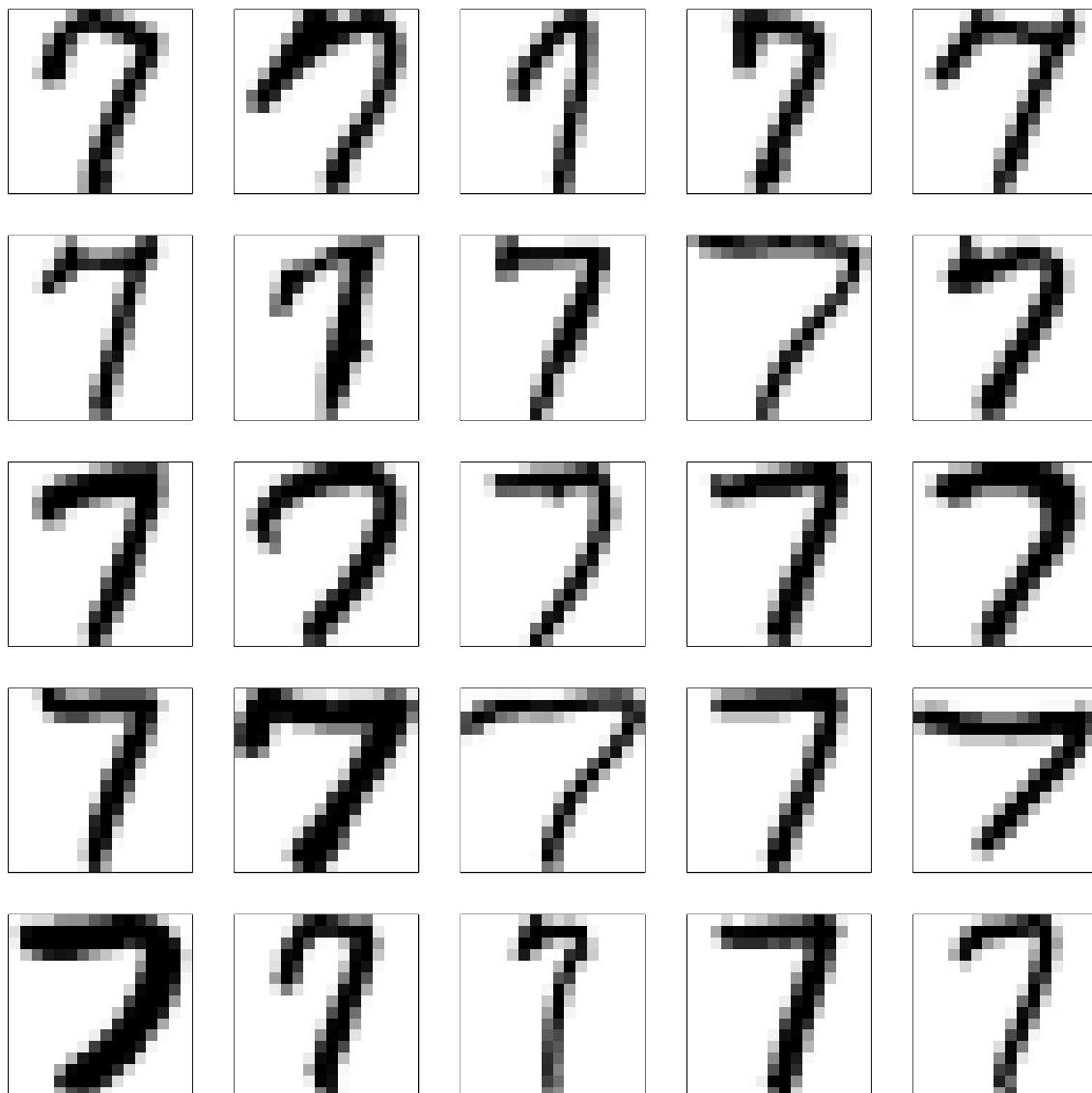


Figure 1.5: The first 25 numeral sevens, digitized

## 1.5 Looking forward

The four examples above illustrate a small sample of the wide variety of data sets that may be encountered. Each of these provides its own challenges. For the baby crawling data, the challenges are more statistical in nature, for example, related to the study design (which isn't described here) and how that design might affect methods of analysis and conclusions drawn from the study. Similar challenges are also present with the other data sets, but the other data sets also present more substantial challenges prior to (and during) the analysis stage, such as how to work with all the missing data in the World Bank data set, or how to effectively and efficiently process the email data to extract features of interest.

This book and associated material introduce, within the context of the R statistical system, tools to tackle some of the challenges in working with real data sets. One important focus is on obtaining and manipulating data. Another is on graphical tools to explore and summarize data. A third focus is on communicating findings about data, and in particular tools that support reproducible research. Tools for classification problems such as email spam filtering or handwritten digit recognition are presented. Programming and writing functions in R are useful skills in all of these focus areas, and will be emphasized.

## 1.6 How to learn (The most important section in this book!)

There are several ways to engage with the content of this book and associated materials.

One way is not to engage at all. Leave the book closed on a shelf and do something else with your time. That may or may not be a good life strategy, depending on what else you do with your time, but you won't learn much from the book!

Another way to engage is to read through the book "passively," reading all that's written but not reading the book while at your computer, where you could enter the R commands from the book. With this strategy you'll probably learn more than if you leave the book closed on a shelf, but there are better options.

A third way to engage is to read the book while you're at a computer, and to enter the R commands from the book as you read about them. You'll likely learn more this way.

A fourth strategy is even better. In addition to reading and entering the commands given in the book, you think about what you're doing, and ask yourself questions (which you then go on to answer). For example after working through the example R code above you might ask yourself, "What would R do if I asked it to calculate the logarithm of a negative number? What would R do if I asked it to calculate the logarithm of a really large number such as one trillion?" You could explore these questions easily, by just trying things out in the R Console window.

If your goal is to maximize the time you have to binge-watch *Stranger Things* on Netflix, the first strategy may be optimal. But if your goal is to learn a lot about computational tools for data science, the fourth strategy is probably going to be best.

# Chapter 2

## Introduction to R and RStudio

Various statistical and programming software environments are used in data science, including R, Python, SAS, C++, SPSS, and many others. Each has strengths and weaknesses, and often two or more are used in a single project. This book focuses on R, for several reasons. One important reason is that R is free. Another is that R is one of, if not the, most widely used software environments in data science. A third is that R is under constant and open development by a diverse and expert core group, and that R also includes an incredible variety of contributed packages. A fourth reason is that a new user can (relatively) quickly gain enough skills to obtain, manage, and analyze data in R.

Several enhanced interfaces for R have been developed. Generally such interfaces are referred to as an integrated development environment (IDE) and are programs to facilitate software development. At minimum, an IDE typically consists of a source code editor and build automation tools. We will use the RStudio IDE, which according to its developers “is a powerful productive user interface for R.”<sup>1</sup> RStudio is widely used, its use is increasing in the R community, and it makes learning to use R a bit simpler. Although we will use RStudio, most of what is presented in this book can be accomplished in R (without an added interface) with few or no changes.

### 2.1 Obtaining and installing R

It is simple to install R on computers running Microsoft Windows, OS X (Mac), and Linux. For other operating systems users can compile the source code directly.<sup>2</sup> Here is a step-by-step guide to installing R for Microsoft Windows.<sup>3</sup> OS X and Linux users would follow similar steps.

---

<sup>1</sup><http://www.rstudio.com/>

<sup>2</sup>Windows, OS X, and Linux users also can compile the source code directly, but for most it is a better idea to install R from already compiled binary distributions.

<sup>3</sup>New versions of R are released regularly, so the version number in Step 6 might be different from what is listed below.

1. Go to <http://www.r-project.org/>
2. Click on the CRAN link on the left side of the page
3. Choose one of the mirrors.<sup>4</sup>
4. Click on Download R for Windows
5. Click on base
6. Click on Download R 3.3.3 for Windows
7. Install R as you would install any other Windows program

## 2.2 Obtaining and installing RStudio

You must install R prior to installing RStudio. RStudio is also simple to install:

1. Go to <http://www.rstudio.com>
2. Click on the link RStudio under the Products tab, then select the Desktop option
3. Click on the Desktop link
4. Choose the DOWNLOAD RSTUDIO DESKTOP link in the Open Source Edition column
5. On the ensuing page, click on the Installer version for your operating system, and once downloaded, install as you would any program

## 2.3 RStudio Server

An alternative to installing R and RStudio on your computer is to use RStudio Server. RStudio Server is a web browser based interface to a current version of R and RStudio. Through this course, you have access to RStudio Server and disk space to store your work and exercises. The only requirements are that you have a fast and reliable internet connection. You can access your RStudio Server account at <http://rstudio.itservices.msu.edu>. Your username is your MSU NetID, and your password is your first name appended to your MSU student ID number (APID), with both the “A” starting your PID and the first letter in your first name capitalized. For example, my username would be *finleya* and my password *A8675309Andrew*<sup>5</sup>.

---

<sup>4</sup>The <http://cran.rstudio.com/> mirror is usually fast. Otherwise choose a mirror in Michigan.

<sup>5</sup>Not surprisingly, this is not really my MSU APID.

This is not a great password because your APID is not private. So, when you first log in, you must change your password to something much more cryptic. Here are the steps to change your password after logging in:

1. Open the `tools` menu at the top
2. Click `Shell...` and a new window will open
3. Type “`passwd`” in the shell window (without the quotes) and press enter
4. Enter current password and press enter
5. Enter new password and press enter
6. Enter your new password again to confirm and press enter
7. You can close the window after you get a confirmation that your password was updated (i.e., the message “`passwd: all authentication tokens updated successfully.`”)

There are advantages and disadvantages to using RStudio Server versus a local install of RStudio on your computer.

#### **Some advantages:**

- You don’t need to install RStudio on your computer
- You don’t need to install the R packages used in the course (we’ve already installed them on the server before the course ends)
- You have plenty of remote (on the cloud) space to save your exercises and related work
- You can access the RStudio Server and course work using any computer with fast and reliable internet

#### **Some disadvantages:**

- You don’t learn firsthand the steps needed to install RStudio and necessary R packages on a computer
- Your access to RStudio Server and course work disappears at the end of the semester (although you can always download a copy of your work)
- It’s a bit cumbersome to move files to and from the server for each exercise
- You can’t do your work offline (you must be connected)
- The server might be slow if there are many users logged in

There are some minor difference between RStudio Server and RStudio that are highlighted in the video at the end of this chapter. Examples given in the remainder of this chapter assume you are using a local install of RStudio (but are applicable to a RStudio Server session).

You are free to use RStudio Server and/or a local install of R/RStudio for this course.

## 2.4 Using R and RStudio

Start RStudio as you would any other program in your operating system. For example under Microsoft Windows it can be run via the Start Menu or by double clicking on the shortcut on the desktop, if a shortcut was created in the installation process. A (rather small) view of RStudio is displayed in Figure 2.1.

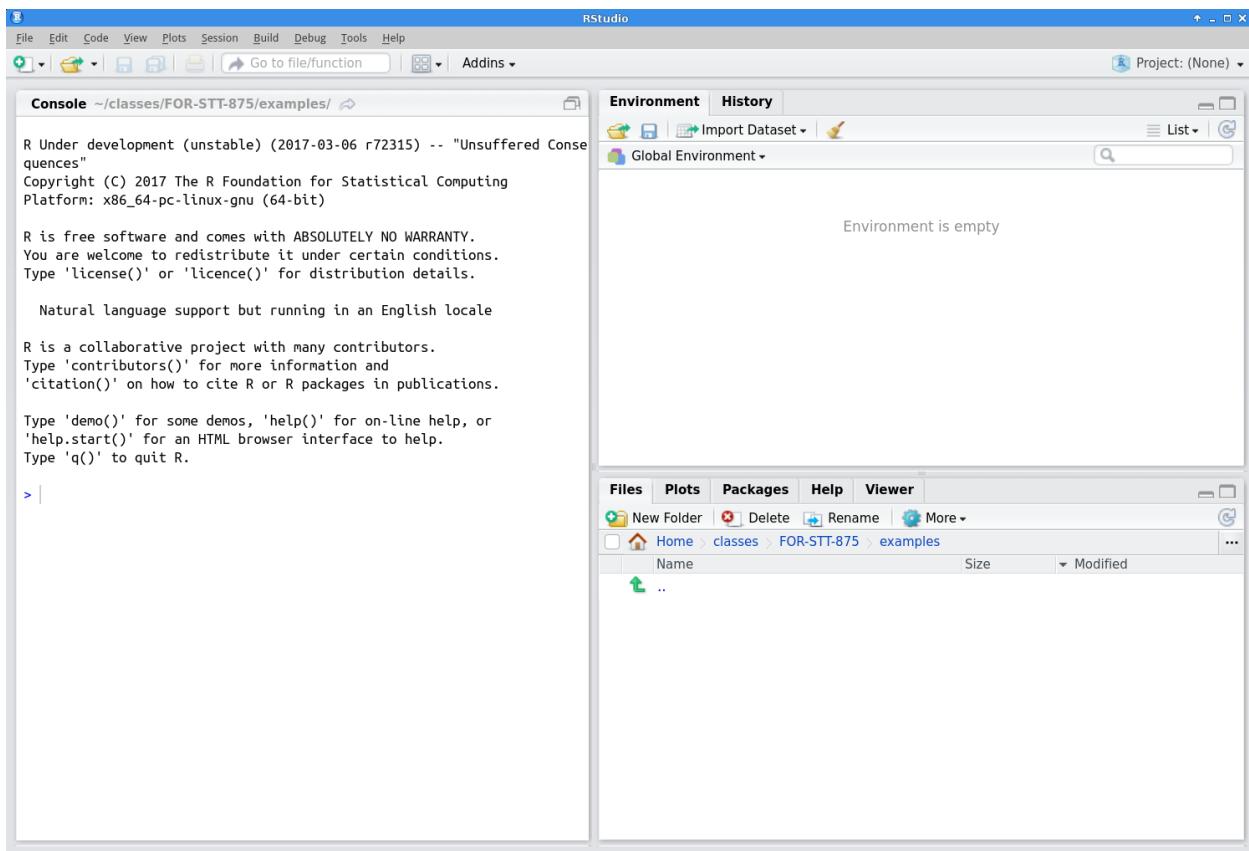


Figure 2.1: The RStudio IDE.

Initially the RStudio window contains three smaller windows. For now our main focus will be the large window on the left, the **Console** window. R statements are typed in this window. The next few sections give simple examples of the use of R. In these sections we will focus on small and non-complex data sets, but of course later in

the book we will work with much larger and more complex sets of data. Read these sections at your computer with R running, and enter the R commands there to get comfortable using the R console window and RStudio.

### 2.4.1 R as a calculator

R can be used as a calculator. Note that # is the comment character in R, so R ignores everything following this character. Also, you will see that R prints [1] before the results of each command. Ignore this for now. Soon its relevance will be explained. The command prompt in R is the greater than sign >.

```
> 34 + 20 * sqrt(100)  ## +,-,*,/ have the expected meanings
[1] 234

> exp(2)  ##The exponential function
[1] 7.389056

> log10(100)  ##Base 10 logarithm
[1] 2

> log(100)  ##Base e logarithm
[1] 4.60517

> 10^log10(55)
[1] 55
```

Most functions in R can be applied to vector arguments rather than operating on a single argument at a time. A vector is a data structure that contains elements of the same data type (i.e. integers).

```
> 1:25  ##The integers from 1 to 25
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25

> log(1:25)  ##The base e logarithm of these integers
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
[6] 1.7917595 1.9459101 2.0794415 2.1972246 2.3025851
[11] 2.3978953 2.4849066 2.5649494 2.6390573 2.7080502
[16] 2.7725887 2.8332133 2.8903718 2.9444390 2.9957323
[21] 3.0445224 3.0910425 3.1354942 3.1780538 3.2188758

> 1:25 * 1:25 ##What will this produce?

[1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196
[15] 225 256 289 324 361 400 441 484 529 576 625

> 1:25 * 1:5 ##What about this?

[1] 1 4 9 16 25 6 14 24 36 50 11 24 39 56
[15] 75 16 34 54 76 100 21 44 69 96 125

> seq(from = 0, to = 1, by = 0.1) ##A sequence of numbers from 0 to 1

[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> exp(seq(from = 0, to = 1, by = 0.1)) ##What will this produce?

[1] 1.000000 1.105171 1.221403 1.349859 1.491825 1.648721
[7] 1.822119 2.013753 2.225541 2.459603 2.718282
```

Now the mysterious square bracketed numbers appearing next to the output make sense. R puts the position of the beginning value on a line in square brackets before the line of output. For example if the output has 40 values, and 15 values appear on each line, then the first line will have [1] at the left, the second line will have [16] to the left, and the third line will have [31] to the left.

## 2.4.2 Basic descriptive statistics and graphics in R

Of course it is easy to compute basic descriptive statistics and to produce standard graphical representations of data. First we create variables with horsepower and miles per gallon. and names for 15 cars.<sup>6</sup> In this case with a small data set we enter the data “by hand” using the `c()` function, which concatenates its arguments into a vector. For larger data sets we will clearly want an alternative. Note that character values are surrounded by quotation marks.

A style note: R has two widely used methods of assignment: the left arrow, which consists of a less than sign followed immediately by a dash: `<-` and the equals sign:

---

<sup>6</sup>These are from a relatively old data set, with 1974 model cars.

- Much ink has been used debating the relative merits of the two methods, and their subtle differences. Many leading R style guides (e.g., the Google style guide at <https://google.github.io/styleguide/Rguide.xml> and the Bioconductor style guide at <http://www.bioconductor.org/developers/how-to/coding-style/>) recommend the left arrow `<-` as an assignment operator, and it will be used below.

Also you will see that if a command has not been completed but the ENTER key is pressed, the command prompt changes to a + sign.

```
> car.hp <- c(110, 110, 93, 110, 175, 105, 245, 62, 95, 123,
+ 123, 180, 180, 180, 205)
> car.mpg <- c(21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8,
+ 19.2, 17.8, 16.4, 17.3, 15.2, 10.4)
> car.name <- c("Mazda RX4", "Mazda RX4 Wag", "Datsun 710",
+ "Hornet 4 Drive", "Hornet Sportabout", "Valiant",
+ "Duster 360", "Merc 240D", "Merc 230", "Merc 280",
+ "Merc 280C", "Merc 450SE", "Merc 450SL",
+ "Merc 450SLC", "Cadillac Fleetwood")
> car.hp
[1] 110 110 93 110 175 105 245 62 95 123 123 180 180 180
[15] 205

> car.mpg
[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8
[12] 16.4 17.3 15.2 10.4

> car.name
[1] "Mazda RX4"           "Mazda RX4 Wag"
[3] "Datsun 710"          "Hornet 4 Drive"
[5] "Hornet Sportabout"   "Valiant"
[7] "Duster 360"          "Merc 240D"
[9] "Merc 230"             "Merc 280"
[11] "Merc 280C"            "Merc 450SE"
[13] "Merc 450SL"           "Merc 450SLC"
[15] "Cadillac Fleetwood"
```

Next we compute some descriptive statistics for the two numeric variables

```
> mean(car.hp)
```

```
[1] 139.7333
```

```
> sd(car.hp)
[1] 50.77607

> summary(car.hp)
   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
 62.0    107.5   123.0   139.7   180.0   245.0

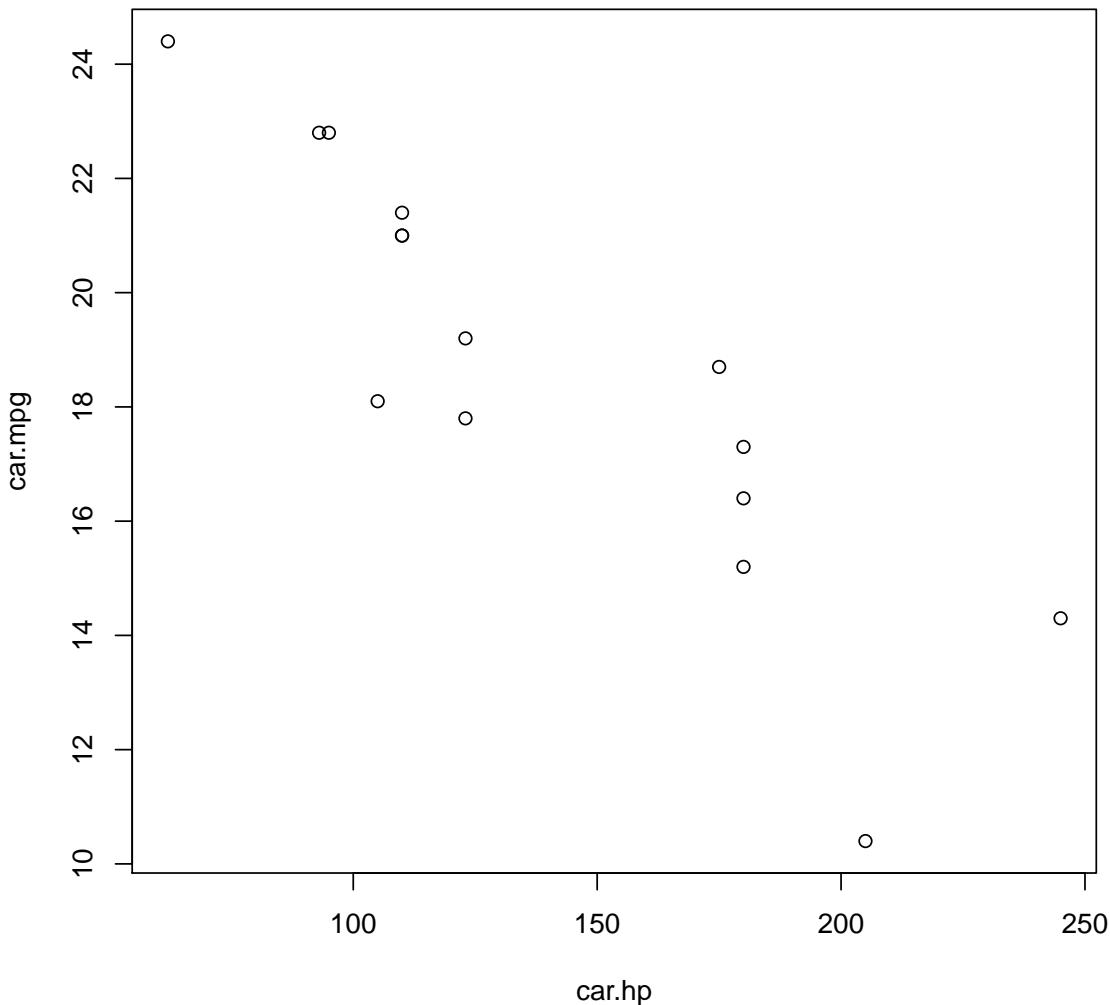
> mean(car.mpg)
[1] 18.72

> sd(car.mpg)
[1] 3.714297

> summary(car.mpg)
   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
10.40   16.85   18.70   18.72   21.20   24.40
```

Next, a scatter plot of `cars.mpg` versus `cars.hp`:

```
> plot(car.hp, car.mpg)
```



Unsurprisingly as horsepower increases, mpg tends to decrease. We'll investigate this further using simple linear regression in the next section.

### 2.4.3 An initial tour of RStudio

When you created the `car.hp` and other vectors in the previous section, you might have noticed the vector name and a short description of its attributes appear in the top right `Global Environment` window. Similarly, when you called `plot(car.hp, car.mpg)` the corresponding plot appeared in the lower right `Plots` window.

A comprehensive, but overwhelming, cheatsheet for RStudio is available here <https://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf>. As we progress in learning R and RStudio, this cheatsheet will become more useful. For now you might use the cheatsheet to locate the various windows and functions identified in

the coming chapters.

## 2.5 How to learn (The most important section in this book!)

There are several ways to engage with the content of this book and associated learning materials.

One way is not to engage at all. Leave the book closed on a shelf and do something else with your time. That may or may not be a good life strategy, depending on what else you do with your time, but you won't learn much from the book!

Another way to engage is to read through the book "passively," reading all that's written but not reading the book while at your computer, where you could enter the R commands from the book. With this strategy you'll probably learn more than if you leave the book closed on a shelf, but there are better options.

A third way to engage is to read the book while you're at a computer, and to enter the R commands from the book as you read about them. You'll likely learn more this way.

A fourth strategy is even better. In addition to reading and entering the commands given in the book, you think about what you're doing, and ask yourself questions (which you then go on to answer). For example after working through the example R code above you might ask yourself, "What would R do if I asked it to calculate the logarithm of a negative number? What would R do if I asked it to calculate the logarithm of a really large number such as one trillion?" You could explore these questions easily, by just trying things out in the R Console window.

If your goal is to maximize the time you have to have to binge-watch Stranger Things on Netflix, the first strategy may be optimal. But if your goal is to learn a lot about computational tools for data science, the fourth strategy is probably going to be best.

## 2.6 Getting help

There are several free (and several not free) ways to get R help when needed.

Several help-related functions are built into R. If there's a particular R function of interest, such as `log`, `help(log)` will bring up a help page for that function. In RStudio the help page is displayed, by default, in the Help tab in the lower right window.<sup>7</sup> The function `help.start` opens a window which allows browsing of the online documentation included with R. To use this, type `help.start()` in the console window.<sup>8</sup> The `help.start` function also provides several manuals online and can be a

---

<sup>7</sup>There are ways to change this default behavior.

<sup>8</sup>You may wonder about the parentheses after `help.start`. A user can specify arguments to any R

quite useful interface to the built in help.

Search engines provide another, sometimes more user-friendly, way to get R questions answered. A Google search often quickly finds something written by another user who had the same (or a similar) question, or an online tutorial that touches on the question. More specialized is [rseek.org](http://rseek.org), which is a search engine focused specifically on R. Both Google and [rseek.org](http://rseek.org) are valuable tools, often providing more user-friendly information than R's own help system.

R users have written many types of contributed documentation. Some of this documentation is available at <http://cran.r-project.org/other-docs.html>. Of course there are also numerous books covering general and specialized R topics available for purchase.

## 2.7 Workspace, working directory, and keeping organized

The *workspace* is your R session working environment and includes any objects you create. Recall these objects are listed in the **Global Environment** window. The command `ls()`, which stands for list, will also list all the objects in your workspace (note, this is the same list that is given in the **Global Environment** window). When you close RStudio, a dialog box will ask you if you want to save an image of the current workspace. If you choose to save your workspace, RStudio saves your session objects and information in a `.RData` file (the period makes it a hidden file) in your *working directory*. Next time you start R or RStudio it checks if there is a `.RData` in the working directory, loads it if it exists, and your session continues where you left off. Otherwise R starts with an empty workspace. This leads to the next question—what is a working directory?

Each R session is associated with a working directory. This is just a directory from which R reads and writes files, e.g., the `.RData` file, data files you want to analyze, or files you want to save. On Mac when you start RStudio it sets the working directory to your home directory (for me that's `/Users/andy`). If you're on a different operating system, you can check where the default working directory is by typing `getwd()` in the console. You can change the default working directory under RStudio's **Global Option** dialog found under the **Tools** dropdown menu. Once an R session is started in RStudio you can change the working directory using options under the **More** button under the **Files** tab in the lower right window. Alternatively, you can set the session's working directory using the `setwd()` in the console. For example, on Windows `setwd("C:/Users/andy/for375/exercise1")` will set the working directory to `C:/Users/andy/for375/exercise1`, assuming that file path and directory exist (note,

---

function inside parentheses. For example `log(10)` asks R to return the logarithm of the argument 10. Even if no arguments are needed, R requires empty parentheses at the end of any function name. In fact if you just type the function name without parentheses, R returns the definition of the function. For simple functions this can be illuminating.

Windows file path uses a backslash, i.e., \, but in R the backslash is an escape character, hence specifying file paths in R on Windows uses the forward slash, i.e., /. Similarly on Mac you can use `setwd("/Users/andy/for375/exercise1")`. Later on when we start reading and writing data from our R session, it will be very important that you are able to identify your current working directory and change it if needed. We will revisit this in subsequent chapters.

As with all work, keeping organized is the key to efficiency. It is good practice to have a dedicated directory for each R project or exercise.

## 2.8 Quality of R code

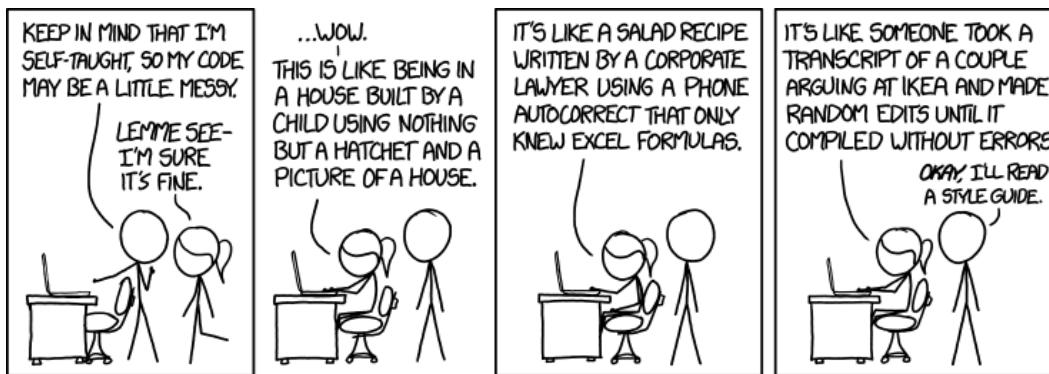


Figure 2.2: xkcd: Code Quality

Writing well-organized and well-labeled code allows your code to be more easily read by another person. (See xkcd's take on code quality in Figure 2.2.) More importantly, though, your well-written code is more accessible to you hours, days, or even months later. We are hoping that you can use the code you write in this class in future projects and research.

Google provides style guides for many programming languages. You can find the R style guide <https://google.github.io/styleguide/Rguide.xml>. Below are a few of the key points from the guide that we will use right away.

### Naming Files

File names should be meaningful and end in .R. If we write a script that analyzes a certain species distribution:

GOOD: `african_rhino_distribution.R`

GOOD: `africanRhinoDistribution.R`

BAD: `speciesDist.R` (too ambiguous)

BAD: `species.dist.R` (too ambiguous and two periods can confuse operating systems' file type auto-detect)

BAD: `speciesdist.R` (too ambiguous and confusing)

### Naming Variables

GOOD: `rhino.count`

GOOD: `rhinoCount`

GOOD: `rhino_count` (We don't mind the underscore and use it quite often, although Google's style guide says it's a no-no for some reason)

Bad: `rhinocount` (confusing)

### Syntax

- Keep code lines under 80 characters long.
- Indent your code with two spaces. (RStudio does this by default when you press the TAB key.)

#### Video 1

This video reviews chapter topics and provides additional details on using the RStudio and RStudio Server IDE.

# Chapter 3

## Scripts, R Markdown, and reproducible research

Doing work in data science, whether for homework, a project for a business, or a research project, typically involves several iterations. For example creating an effective graphical representation of data can involve trying out several different graphical representations, and then tens if not hundreds of iterations when fine-tuning the chosen representation. And each of these representations may require several R commands to create. Although this all could be accomplished by typing and re-typing commands at the R Console, it is easier and more effective to write the commands in a *script file* which then can be submitted to the R console either a line at a time or all together.<sup>1</sup>

In addition to making the workflow more efficient, R scripts provide another large benefit. Often we work on one part of a homework assignment or project for a few hours, then move on to something else, and then return to the original part a few days, months, or sometimes even years later. In such cases we may have forgotten how we created the graphical display, for example, that we were so proud of, and will need to again spend a few hours to recreate it. If we save a script file, we have the ingredients immediately available when we return to a portion of a project.<sup>2</sup>

Next consider the larger scientific endeavor. Ideally a scientific study will be reproducible, meaning that an independent group of researchers (or the original researchers) will be able to duplicate the study. Thinking about data science, this means that all the steps that were taken when working with the data from a study should be reproducible, from selection of variables to formal data analysis. In principle this can be facilitated by explaining, in words, each step of the work with data. In practice it is typically difficult or impossible to reproduce a full data analysis based on a written explanation. Much more effective is to include the actual computer code which accomplished the data work in the report, whether the report is a homework assignment or a research

---

<sup>1</sup>Unsurprisingly it is also possible to submit several selected lines of code at once.

<sup>2</sup>In principle the R history mechanism provides a similar record. But with history we have to search through a lot of other code to find what we're looking for, and scripts are a much cleaner mechanism to record our work.

paper. Tools in R such as *R Markdown* facilitate this process.

## 3.1 Scripts in R

As noted above, scripts help to make work with data more efficient and provide a record of how data were managed and analyzed. Here is an example. This example uses features of R that we have not yet discussed, so don't worry about the details, but rather about how it motivates the use of a script file.

First we read in a data set containing data on (among other things) fertility rate and life expectancy for countries of the world, for the years 1960 through 2014.

```
> u <- "http://blue.for.msu.edu/FOR875/data/WorldBank.csv"
> WorldBank <- read.csv(u, header = TRUE, stringsAsFactors = FALSE)
```

Next we print the names of the variables in the data set. Don't be concerned about the specific details. Later we will learn much more about reading in data and working with data sets in R.

```
> names(WorldBank)

[1] "iso2c"
[2] "country"
[3] "year"
[4] "fertility.rate"
[5] "life.expectancy"
[6] "population"
[7] "GDP.per.capita.Current.USD"
[8] "X15.to.25.yr.female.literacy"
[9] "iso3c"
[10] "region"
[11] "capital"
[12] "longitude"
[13] "latitude"
[14] "income"
[15] "lending"
```

We will try to create a scatter plot of fertility rate versus life expectancy of countries for the year 1960. To do this we'll first create variables containing the values of fertility rate and life expectancy for 1960<sup>3</sup>, and print out the first ten values of each variable.

---

<sup>3</sup>This isn't necessary, but it is convenient

```
> fertility <- WorldBank$fertility.rate[WorldBank$year == 1960]
> lifeexp <- WorldBank$life.expectancy[WorldBank$year == 1960]
> fertility[1:10]

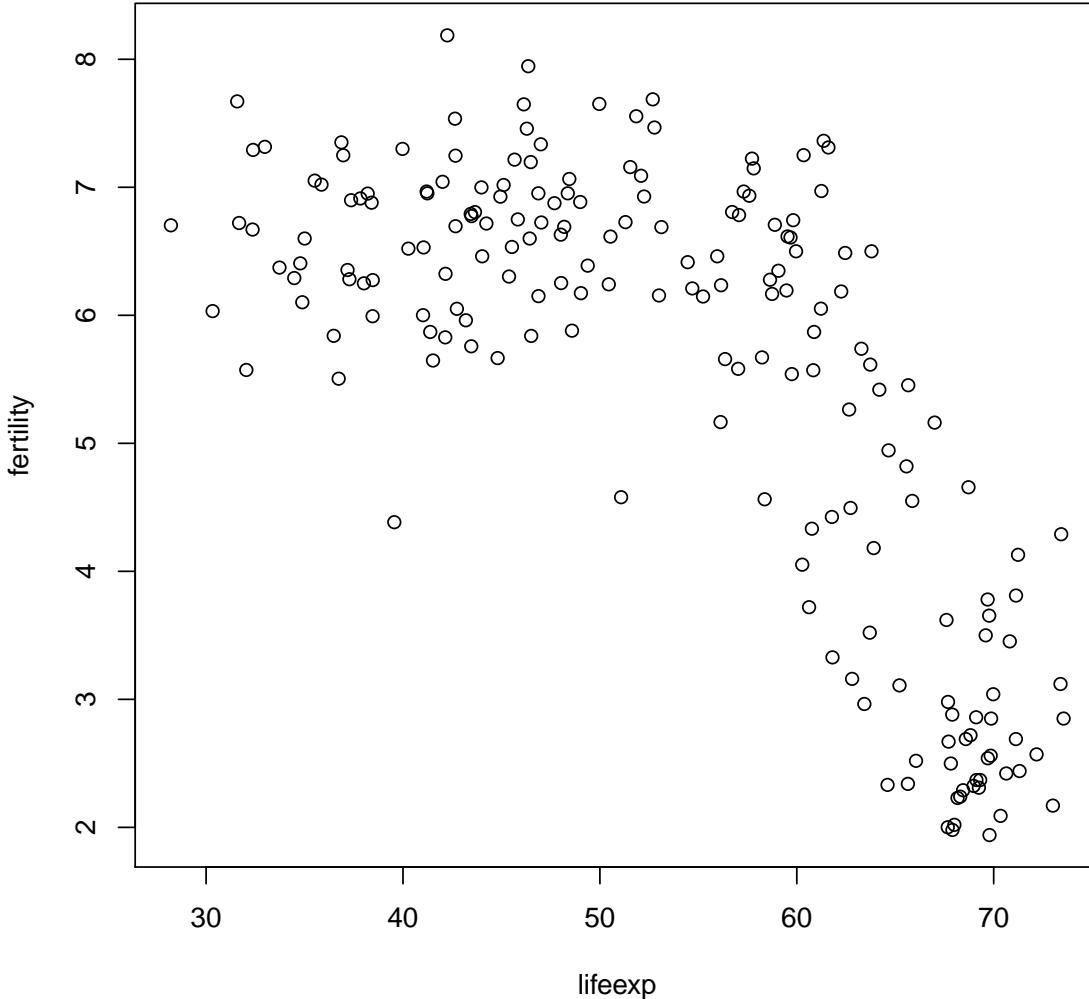
[1]      NA 6.928 7.671 4.425 6.186 4.550 7.316 3.109      NA
[10] 2.690

> lifeexp[1:10]

[1]      NA 52.24322 31.58005 61.78273 62.25437 65.86346
[7] 32.98483 65.21554      NA 68.58561
```

We see that some countries do not have data for 1960. R represents missing data via `NA`. Of course at some point it would be good to investigate which countries' data are missing and why. The `plot()` function in R will just omit missing values, and for now we will just plot the non-missing data. A scatter plot of the data is drawn next.

```
> plot(lifeexp, fertility)
```



The scatter plot shows that as life expectancy increases, fertility rate tends to decrease, and the relationship seems nonlinear. Now that we have a basic scatter plot, it is tempting to make it more informative. We will do this by adding two features. One is to make the points' size proportional to the country's population, and the second is to make the points' color represent the region of the world the country resides in. We'll first extract the population and region variables for 1960.

```
> pop <- WorldBank$population[WorldBank$year == 1960]
> region <- WorldBank$region[WorldBank$year == 1960]
> pop[1:10]
```

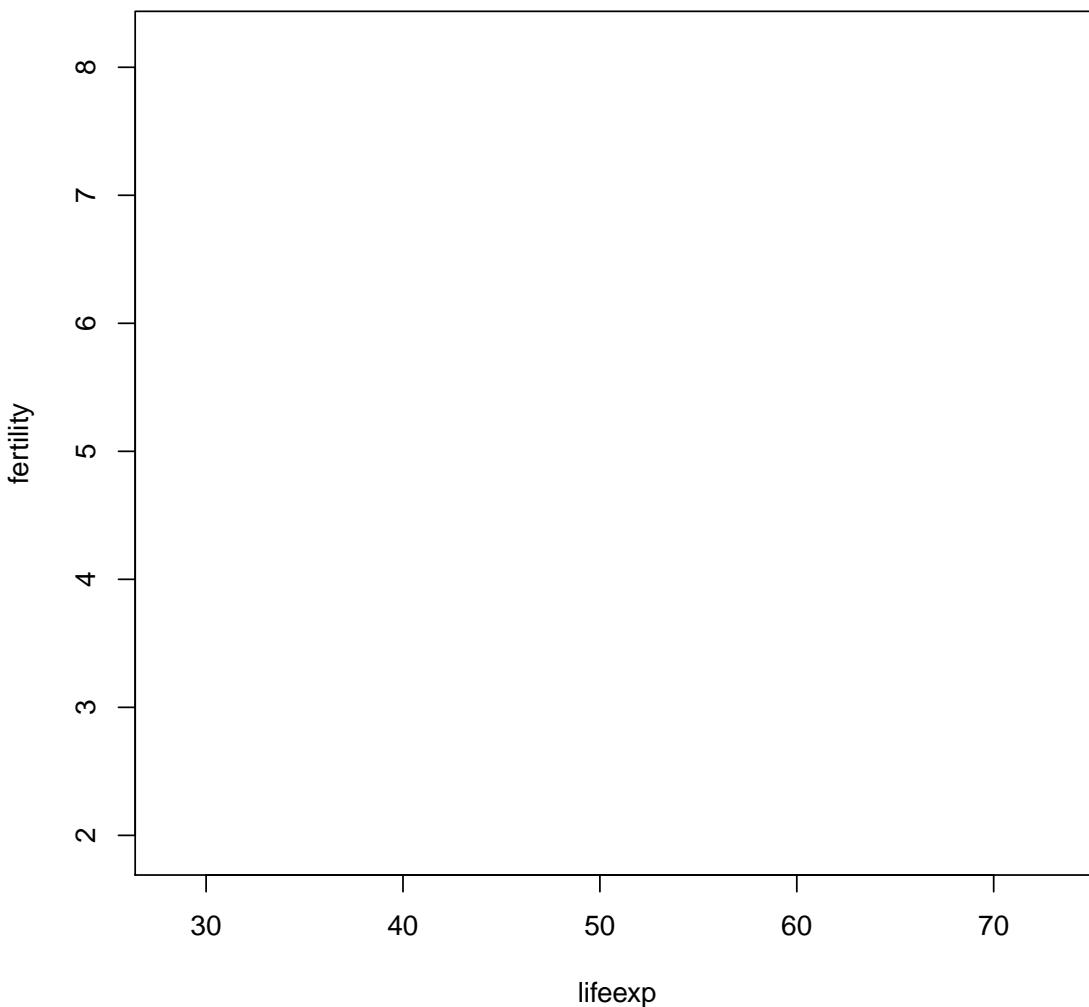
[1]	13414	89608	8774440	54681	1608800	1867396
[7]	4965988	20623998	20012	7047539		

```
> region[1:10]

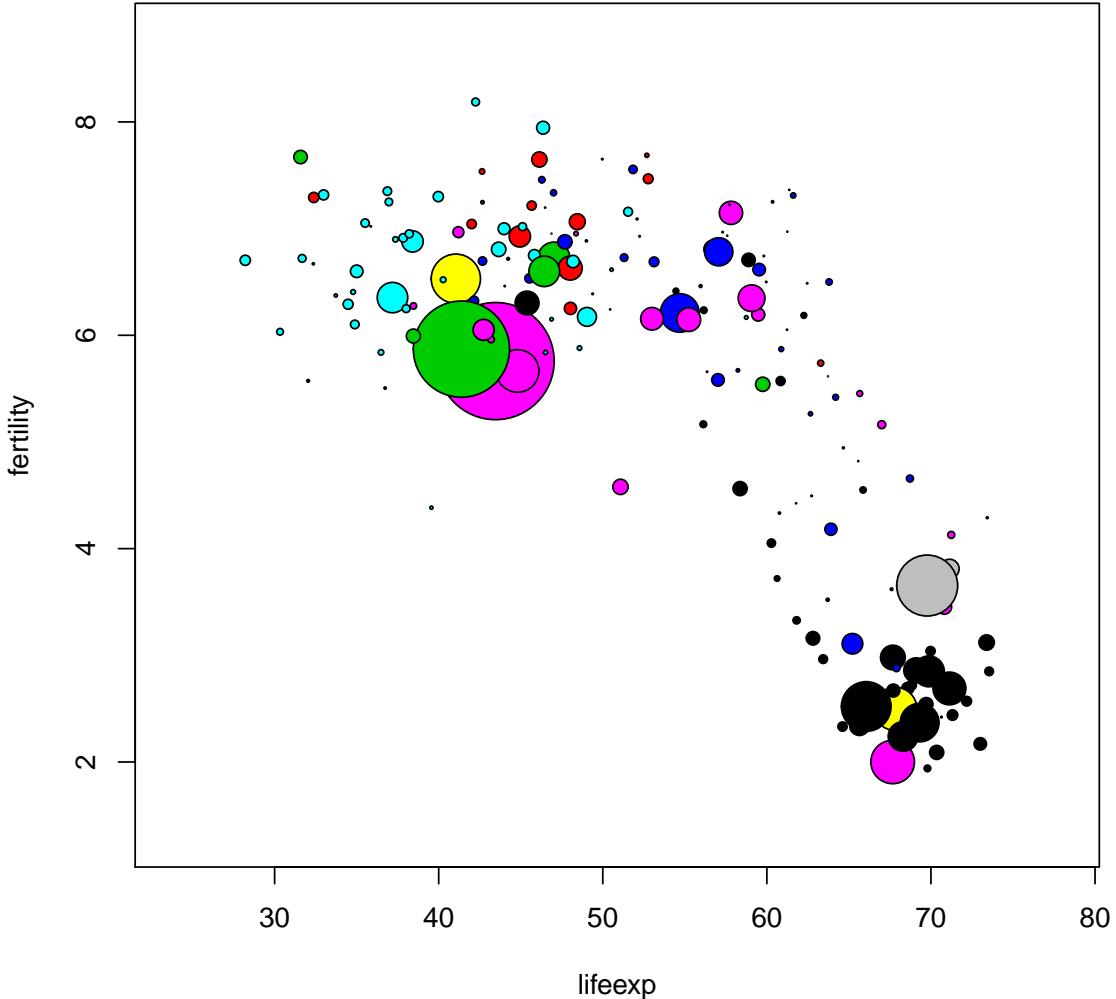
[1] "Europe & Central Asia (all income levels)"
[2] "Middle East & North Africa (all income levels)"
[3] "South Asia"
[4] "Latin America & Caribbean (all income levels)"
[5] "Europe & Central Asia (all income levels)"
[6] "Europe & Central Asia (all income levels)"
[7] "Sub-Saharan Africa (all income levels)"
[8] "Latin America & Caribbean (all income levels)"
[9] "East Asia & Pacific (all income levels)"
[10] "Europe & Central Asia (all income levels)"
```

To create the scatter plot we will do two things. First we will create the axes, etc. for the plot, but not plot the points. The argument `type="n"` tells R to do this. Then we use the `symbols()` function to add symbols, and use the `circles` argument to set the sizes of the points, and the `bg` argument to set the colors. Don't worry about the details! In fact later in the book we will learn about an R package called `ggplot2` that provides a simpler way to create such plots. You'll see two plots below, first the "empty" plot which is just a building block, then the plot including the appropriate symbols.

```
> plot(lifeexp, fertility, type="n")
```



```
> symbols(lifeexp, fertility, circles=sqrt(pop/pi), inches=0.35,  
+           bg=match(region, unique(region)))
```



Of course we should have a key which tells the viewer which region each color represents, and a way to determine which country each point represents, and a lot of other refinements. For now we will resist such temptations.

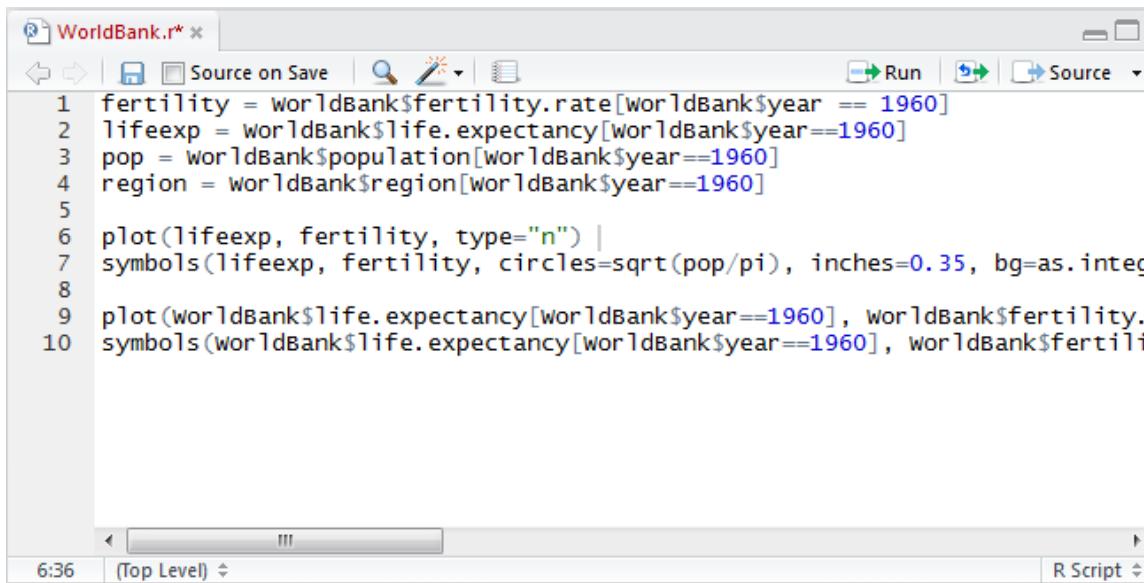
Some of the process leading to the completed plot is shown above, such as reading in the data, creating variables representing the 1960 fertility rate, life expectancy, an intermediate plot that was rejected, and so on. A lot of the process isn't shown, to save space. There would likely be mistakes (either minor typing mistakes or more complex errors), for example. Focusing only on the `symbols()` function that was used to add the colorful symbols to the scatter plot, there would likely have been a substantial number of attempts with different values of the `circles`, `inches`, and `bg` arguments before settling on the actual form used to create the plot.

Now imagine trying to recreate the plot a few days later. Possibly someone saw the plot and commented that it would be interesting to see some similar plots, but for

years in the 1970s when there were major famines in different countries of the world. If all the work, including all the false starts and refinements, were done at the console, it would be hard to sort things out and would take longer than necessary to create the new plots. This would be especially true if a few months rather than a few days had passed.

But with a script file, especially a script file with a few well-chosen comments, creating the new scatter plots would be much easier. Fortunately it is quite easy to create and work with script files in RStudio.<sup>4</sup> Just choose **File > New File > New script** and a script window will open up in the upper left of the full RStudio window.

An example of a script window (with some R code already typed in) is shown in Figure 3.1. From the script window the user can, among other things, save the script (either using the **File** menu or the icon near the top left of the window) and can run one or more lines of code from the window (using the **run** icon in the window, or by copying and pasting into the console window). In addition, there is a **Source on Save** checkbox. If this is checked, the R code in the script window is automatically read into R and executed when the script file is saved.



```
fertility = worldBank$fertility.rate[worldBank$year == 1960]
lifeexp = worldBank$life.expectancy[worldBank$year==1960]
pop = Wor1dBank$population[wor1dBANK$year==1960]
region = wor1dBANK$region[wor1dBANK$year==1960]
plot(lifeexp, fertility, type="n")
symbols(lifeexp, fertility, circles=sqrt(pop/pi), inches=0.35, bg=as.integer(region))
plot(worldBank$life.expectancy[worldBank$year==1960], worldBank$fertility)
symbols(worldBank$life.expectancy[worldBank$year==1960], worldBank$fertility)
```

Figure 3.1: A script window in RStudio.

## 3.2 R Markdown

People typically work on data with a larger purpose in mind. Possibly the purpose is to understand a biological system more clearly. Possibly the purpose is to refine a system that recommends movies to users in an online streaming movie service. Possibly the

---

<sup>4</sup>It is also easy in R without RStudio. Just use **File > New script** to create a script file, and save it before exiting R.

purpose is to complete a homework assignment and demonstrate to the instructor an understanding of an aspect of data analysis. Whatever the purpose, a key aspect is communicating with the desired audience, for example fellow researchers or an instructor.

One possibility, which is somewhat effective, is to write a document using software such as Microsoft Word<sup>5</sup> and to include R output such as computations and graphics by cutting and pasting into the main document. One drawback to this approach is similar to what makes script files so useful: If the document must be revised it may be hard to unearth the R code that created graphics or analyses, to revise these.<sup>6</sup> A more subtle but possibly more important drawback is that the reader of the document will not know precisely how analyses were done, or how graphics were created. And over time even the author(s) of the paper will forget the details. A verbal description in a “methods” section of a paper can help here, but typically these do not provide all the details of the analysis, but rather might state something like, “All analyses were carried out using R version 3.3.1.”

RStudio’s website provides an excellent overview of R Markdown capabilities for reproducible research. At minimum, follow the `Get Started` link at <http://rmarkdown.rstudio.com/> and watch the introduction video.

Among other things, R Markdown provides a way to include R code that read in data, create graphics, or perform analyses, in a single document which is processed to create a research paper or homework assignment or other written product. The R Markdown file is a plain text file containing text the author wants to have shown in the final document, simple commands to indicate how the text should be formatted (for example boldface or italic, or a bulleted list), and R code which creates output (including graphics) on the fly. Perhaps the simplest way to get started is to see an R Markdown file, and the resulting document that is produced after the R Markdown document is processed. Figure 3.2 shows a very simple R Markdown file, and Figure 3.3 shows the resulting output. In this case the output created is an HTML file, but there are other possible output formats such as Microsoft Word or PDF.

At the top of the input R Markdown file are some lines with `---` at the top and the bottom. These lines are not needed, but give a convenient way to specify the title, author, and date of the article, that are then typeset prominently at the top of the output document. For now, don’t be concerned with the lines following `output::`. These can be omitted (or included as shown).

Next are a few lines showing some of the ways that font effects such as italics, boldface, and strikethrough can be achieved. For example, an asterisk before and after text sets the text in *italics*, and two asterisks before and after text sets the text in **boldface**.

More important for our purposes is the ability to include R code in the R Markdown

---

<sup>5</sup>Or possibly L<sup>A</sup>T<sub>E</sub>X if the document is more technical

<sup>6</sup>Organizing the R code using script files and keeping all the work organized in a well-thought-out directory structure can help here, but this requires a level of forethought and organization that most people do not possess...including myself.

file, which will be executed with the output appearing in the output document. Bits of R code included this way are called *code chunks*. The beginning of a code chunk is indicated with three backticks and an “r” in curly braces: ` ` ` {r}. The end of a code chunk is indicated with three backticks: ` ` ` . For example, the R Markdown file in Figure 3.2 has one code chunk:

```
```{r}
x <- 1:10
y <- 10:1
mean(x)
sd(y)
```

```

In this code chunk two vectors `x` and `y` are created, and the mean of `x` and the standard deviation of `y` are computed. In the output in Figure 3.3 the R code is reproduced, and the output of the two lines of code asking for the mean and standard deviation is shown.

### 3.2.1 Creating and processing R Markdown documents

RStudio has features which facilitate creating and processing R Markdown documents. Choose `File > New File > R Markdown....`. In the ensuing dialog box make sure that `Document` is highlighted on the left, enter the title and author (if desired), and choose the Default Output Format (HTML is good to begin). Then click OK. A document will appear in the upper left of the RStudio window. It is an R Markdown document, and the title and author you chose will show up, delimited by --- at the top of the document. A generic body of the document will also be included.

For now just keep this generic document as is. To process it to create the HTML output, click the `Knit HTML` button at the top of the R Markdown window. You’ll be prompted to choose a filename for the R Markdown file. Use `.Rmd` as the extension for this file. Once you’ve saved the file, RStudio will process the file, create the HTML output, and open this output in a new window. The HTML output file will also be saved to your working directory. This file can be shared with others, who can open it using a web browser such as Chrome or Firefox.

There are many options which allow customization of R Markdown documents. Some of these affect formatting of text in the document, while others affect how R code is evaluated and displayed. The RStudio web site contains a useful summary of many R Markdown options at [www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf](http://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf). A different, but mind-numbingly busy, cheatsheet is at [www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf](http://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf). Some of the more commonly used R Markdown options are described next.

```
---
title: "R Markdown"
author: "Andy Finley"
date: "April 3, 2017"
output: html_document
---
```

Basic formatting:

**\*italic\***

**\*\*bold\*\***

**~~strikethrough~~**

A code chunk:

```
```{r}
x <- 1:10
y <- 10:1
mean(x)
sd(y)
```
```

Inline code:

``r 5+5``

Inline code not executed:

``5+5``

Figure 3.2: R Markdown file which produces the output in Figure 3.3.

# R Markdown

*Andy Finley*

*April 3, 2017*

Basic formatting:

*italic*

**bold**

~~strikethrough~~

A code chunk:

```
x <- 1:10
y <- 10:1
mean(x)
```

```
## [1] 5.5
```

```
sd(y)
```

```
## [1] 3.02765
```

Inline code:

10

Inline code not executed:

5+5

Figure 3.3: Output from the R Markdown file in Figure 3.2.

## Text: Lists and Headers

Unordered (sometimes called bulleted) lists and ordered lists are easy in R Markdown. Figures 3.4 and 3.5 illustrate the creation of unordered and ordered lists.

- For an unordered list, either an asterisk, a plus sign, or a minus sign may precede list items. Use a space after these symbols before including the list text. To have second-level items (sub-lists) indent four spaces before indicating the list item. This can also be done for third-level items.

- For an ordered list use a numeral followed by a period and a space (1. or 2. or 3. or ...) to indicate a numbered list, and use a letter followed by a period and a space (a. or b. or c. or ...) to indicate a lettered list. The same four space convention is used to designate sub lists.
- For an ordered list, the first list item will be labeled with the number or letter that you specify, but subsequent list items will be numbered sequentially. The example in Figures 3.4 and 3.5 will make this more clear. In those examples notice that for the ordered list, although the first-level numbers given in the R Markdown file are 1, 2, and 17, the numbers printed in the output are 1, 2, and 3. Similarly the letters given in the R Markdown file are c and q, but the output file prints c and d.

An unordered list:

```
* List item 1
* List item 2
  + Second level list item 1
  + Second level list item 2
    + Third level list item
* List item 3
```

An ordered list:

```
1. List item 1
2. List item 2
  c. Sub list item 1
  q. Sub list item 2
17. List item 3
```

An unordered list:

```
• List item 1
• List item 2
  – Second level list item 1
  – Second level list item 2
    * Third level list item
• List item 3
```

An ordered list:

```
1. List item 1
2. List item 2
  c. Sub list item 1
  d. Sub list item 2
3. List item 3
```

Figure 3.4: R Markdown file to produce unordered and ordered lists.

Figure 3.5: Output from the R Markdown file in Figure 3.4.

R Markdown does not give substantial control over font size. Different “header” levels, which provide different font sizes, are available. Put one or more hash marks in front of text to specify different header levels. Other font choices such as subscripts and superscripts are possible, by surrounding the text either by tildes or carets. More sophisticated mathematical displays are also possible, and are surrounded by dollar signs. The actual mathematical expressions are specified using a language called L<sup>A</sup>T<sub>E</sub>X. See Figures 3.6, 3.7, and 3.8 for examples.

```
# A first *level* ~~header~~
## A second level header
### A third level header
```

Text subscripts and superscripts:

$x^{2^{\wedge}} + y^{2^{\wedge}}$

$10^{3^{\wedge}} = 1000$

Mathematics examples:

$\$x\_a\$$

$\$x^a\$$

$\$\backslash int\_0^1 x^2 dx\$$

$\$\backslash frac{x}{y}\$$

$\$\backslash sqrt{x}\$$

$\$\backslash sqrt[n]{x}\$$

$\$\backslash sum_{k=1}^n\$

$\$\backslash prod_{k=1}^n\$

## A first *level* header

### A second level header

#### A third level header

Text subscripts and superscripts:

$x_2 + y_2$

$10^3 = 1000$

Mathematics examples:

$x_a$

$x^a$

$\int_0^1 x^2 dx$

$\frac{x}{y}$

$\sqrt{x}$

$\sqrt[n]{x}$

$\sum_{k=1}^n$

$\prod_{k=1}^n$

Figure 3.6: R Markdown which produces the output in Figure 3.7.

Figure 3.7: Output from the R Markdown file in Figure 3.6.

## Code Chunks

R Markdown provides a large number of options to vary the behavior of code chunks. In some contexts it is useful to display the output but not the R code leading to the output. In some contexts it is useful to display the R prompt and in others it is not. Possibly the size of figures created by graphics commands should be configurable. And so on. A large number of code chunk options are described in [www.rstudio.com/  
wp-content/uploads/2015/03/rmarkdown-reference.pdf](http://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf).

Code chunk options are specified in the curly braces near the beginning of a code chunk. For example the option `echo=FALSE` would be specified via ````{r, echo=FALSE}`.

|            |                       |           |                      |           |                      |                   |                              |
|------------|-----------------------|-----------|----------------------|-----------|----------------------|-------------------|------------------------------|
| $\leq$     | <code>\leq</code>     | $\geq$    | <code>\geq</code>    | $\neq$    | <code>\neq</code>    | $\approx$         | <code>\approx</code>         |
| $\times$   | <code>\times</code>   | $\div$    | <code>\div</code>    | $\pm$     | <code>\pm</code>     | $\cdot$           | <code>\cdot</code>           |
| $\circ$    | <code>\circ</code>    | $\circ$   | <code>\circ</code>   | $/$       | <code>\prime</code>  | $\dots$           | <code>\cdots</code>          |
| $\infty$   | <code>\infty</code>   | $\neg$    | <code>\neg</code>    | $\wedge$  | <code>\wedge</code>  | $\vee$            | <code>\vee</code>            |
| $\supset$  | <code>\supset</code>  | $\forall$ | <code>\forall</code> | $\in$     | <code>\in</code>     | $\rightarrow$     | <code>\rightarrow</code>     |
| $\subset$  | <code>\subset</code>  | $\exists$ | <code>\exists</code> | $\notin$  | <code>\notin</code>  | $\Rightarrow$     | <code>\Rightarrow</code>     |
| $\cup$     | <code>\cup</code>     | $\cap$    | <code>\cap</code>    | $ $       | <code>\mid</code>    | $\Leftrightarrow$ | <code>\Leftrightarrow</code> |
| $\dot{a}$  | <code>\dot{a}</code>  | $\hat{a}$ | <code>\hat{a}</code> | $\bar{a}$ | <code>\bar{a}</code> | $\tilde{a}$       | <code>\tilde{a}</code>       |
| $\alpha$   | <code>\alpha</code>   | $\beta$   | <code>\beta</code>   | $\gamma$  | <code>\gamma</code>  | $\delta$          | <code>\delta</code>          |
| $\epsilon$ | <code>\epsilon</code> | $\zeta$   | <code>\zeta</code>   | $\eta$    | <code>\eta</code>    | $\varepsilon$     | <code>\varepsilon</code>     |
| $\theta$   | <code>\theta</code>   | $\iota$   | <code>\iota</code>   | $\kappa$  | <code>\kappa</code>  | $\vartheta$       | <code>\vartheta</code>       |
| $\lambda$  | <code>\lambda</code>  | $\mu$     | <code>\mu</code>     | $\nu$     | <code>\nu</code>     | $\xi$             | <code>\xi</code>             |
| $\pi$      | <code>\pi</code>      | $\rho$    | <code>\rho</code>    | $\sigma$  | <code>\sigma</code>  | $\tau$            | <code>\tau</code>            |
| $\upsilon$ | <code>\upsilon</code> | $\phi$    | <code>\phi</code>    | $\chi$    | <code>\chi</code>    | $\psi$            | <code>\psi</code>            |
| $\omega$   | <code>\omega</code>   | $\Gamma$  | <code>\Gamma</code>  | $\Delta$  | <code>\Delta</code>  | $\Theta$          | <code>\Theta</code>          |
| $\Lambda$  | <code>\Lambda</code>  | $\Xi$     | <code>\Xi</code>     | $\Pi$     | <code>\Pi</code>     | $\Sigma$          | <code>\Sigma</code>          |
| $\Upsilon$ | <code>\Upsilon</code> | $\Phi$    | <code>\Phi</code>    | $\Psi$    | <code>\Psi</code>    | $\Omega$          | <code>\Omega</code>          |

Figure 3.8: Other useful L<sup>A</sup>T<sub>E</sub>X expressions and symbols available for use in R Markdown.

Below a few of the more commonly used options are described. The use of these options is illustrated in Figures 3.9 and 3.10.

1. `echo=FALSE` specifies that the R code should not be printed (but any output of the R code should be printed) in the resulting document.
2. `include=FALSE` specifies that neither the R code nor the output should be printed. However, the objects created by the code chunk will be available for use in later code chunks.
3. `eval=FALSE` specifies that the R code should not be evaluated. The code will be printed unless, for example, `echo=FALSE` is also given as an option.
4. `error=FALSE` and `warning=FALSE` specify that, respectively, error messages and warning messages generated by the R code should not be printed.
5. The `comment` option allows a specified character string to be prepended to each line of results. By default this is set to `comment = '##'` which explains the two hash marks preceding results in Figure 3.3 for example. Setting `comment = NA` presents output without any character string prepended. That is done in most code chunks in this book.
6. `prompt=TRUE` specifies that R prompt `>` will be prepended to each line of R code shown in the document. `prompt = FALSE` specifies that command prompts should not be included.

7. `fig.height` and `fig.width` specify the height and width of figures generated by R code. These are specified in inches, so for example `fig.height=4` specifies a four inch high figure.

Figures 3.9 (R Markdown input) and 3.10 (printed output) give examples of the use of code chunk options.

```
No options:  
```{r}  
x <- 1:10  
x  
```  
  
echo=FALSE:  
```{r, echo=FALSE}  
x <- 1:10  
x  
```  
  
comment=NA:  
```{r, comment=NA}  
x <- 1:10  
x  
```  
  
comment='#', prompt=TRUE:  
```{r, comment='#', prompt=TRUE}  
x <- 1:10  
x  
```  
  
echo=FALSE, fig.height=4, fig.width=4:  
  
```{r, echo=FALSE, fig.height=4, fig.width=4}  
y <- 10:1  
plot(x,y)  
```
```

Figure 3.9: R Markdown illustrating some of the options for code chunks.

No options:

```
x = 1:10
x
## [1] 1 2 3 4 5 6 7 8 9 10
```

echo=FALSE:

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

comment=NA:

```
x = 1:10
x
[1] 1 2 3 4 5 6 7 8 9 10
```

comment='#', prompt=TRUE:

```
> x = 1:10
> x
# [1] 1 2 3 4 5 6 7 8 9 10
```

echo=FALSE, fig.height=4, fig.width=4:

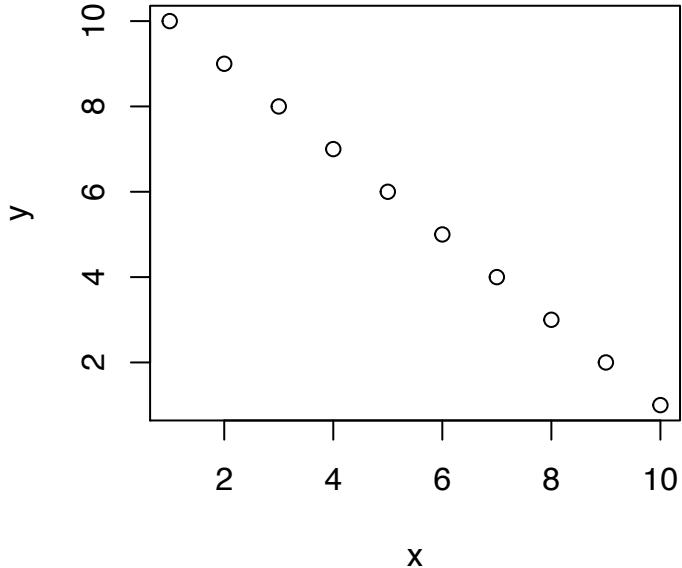


Figure 3.10: Output from the R Markdown in Figure 3.9.

### 3.2.2 Output formats other than HTML

It is possible to use R Markdown to produce documents in formats other than HTML, including Word and PDF documents, among others. Next to the `Knit HTML` button is a down arrow. Click on this and choose `Knit Word` to produce a Microsoft word output document. Although there is also a `Knit PDF` button, PDF output requires additional software called `TeX` in addition to RStudio.<sup>7</sup>

### 3.2.3 L<sup>A</sup>T<sub>E</sub>X and knitr

While R Markdown provides substantial flexibility and power, it lacks features such as cross-referencing, fine control over fonts, etc. If this is desired, a variant of R Markdown called `knitr`, which has very similar syntax to R Markdown for code chunks, can be used in conjunction with the typesetting system L<sup>A</sup>T<sub>E</sub>X to produce documents. In fact this book was created using `knitr` and L<sup>A</sup>T<sub>E</sub>X. For simpler tasks, however, R Markdown is sufficient, and substantially easier to learn.

#### Exercise 1

Learning objectives: practice setting up a working directory and read in data; explore the workspace within RStudio and associated commands; produce basic descriptive statistics and graphics.

#### Exercise 2

Learning objectives: practice working within RStudio; create a R Markdown document and resulting html document in RStudio; calculate descriptive statistics and graphics.

---

<sup>7</sup>It isn't particularly hard to install `TeX` software. For a Microsoft Windows system, MiKTeX is convenient, and is available from [miktex.org](http://miktex.org). For a Mac system, MacTeX is available from [www.tug.org/mactex/](http://www.tug.org/mactex/).

# Chapter 4

## Data Structures

A data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter's focus is on data structures. To help initial understanding, the data in this chapter will be relatively modest in size and complexity. The ideas and methods, however, generalize to larger and more complex data sets.

The base data structures in R are vectors, matrices, arrays, data frames, and lists. The first three, vectors, matrices, and arrays, require all elements to be of the same type or homogeneous, e.g., all numeric or all character. Data frames and lists allow elements to be of different types or heterogeneous, e.g., some elements of a data frame may be numeric while other elements may be character. These base structures can also be organized by their dimensionality, i.e., 1-dimensional, 2-dimensional, or N-dimensional, as shown in Table 4.1.

| Dimension | Homogeneous   | Heterogeneous |
|-----------|---------------|---------------|
| 1         | Atomic vector | List          |
| 2         | Matrix        | Data frame    |
| N         | Array         |               |

Table 4.1: Dimension and type content of base data structures in R.

R has no scalar types, i.e., 0-dimensional. Individual numbers or strings are actually vectors of length one.

An efficient way to understand what comprises a given object is to use the `str()` function. `str()` is short for structure and prints a compact, human-readable description of any R data structure. For example, in the code below, we prove to ourselves that what we might think of as a scalar value is actually a vector of length one.

```
> a <- 1
> str(a)

num 1

> is.vector(a)

[1] TRUE

> length(a)

[1] 1
```

Here we assigned `a` the scalar value one. The `str(a)` prints `num 1`, which says `a` is numeric of length one. Then just to be sure we used the function `is.vector()` to test if `a` is in fact a vector. Then, just for fun, we asked the length of `a`, which again returns one. There are a set of similar logical tests for the other base data structures, e.g., `is.matrix()`, `is.array()`, `is.data.frame()`, and `is.list()`, these will all come in handy as we encounter different R objects.

## 4.1 Vectors

Think of a vector<sup>1</sup> as a structure to represent one variable in a data set. For example a vector might hold the weights, in pounds, of 7 people in a data set. Or another vector might hold the genders of those 7 people. The `c()` function in R is useful for creating (small) vectors and for modifying existing vectors. Think of `c` as standing for “combine.”

```
> weight <- c(123, 157, 205, 199, 223, 140, 105)
> weight

[1] 123 157 205 199 223 140 105

> gender <- c("female", "female", "male", "female", "male",
+             "male", "female")
> gender

[1] "female" "female" "male"    "female" "male"    "male"
[7] "female"
```

---

<sup>1</sup>Technically the objects described in this section are “atomic” vectors (all elements of the same type), since lists, to be described below, also are actually vectors. This will not be an important issue, and the shorter term vector will be used for atomic vectors below.

Notice that elements of a vector are separated by commas when using the `c()` function to create a vector. Also notice that character values are placed inside quotation marks.

The `c()` function also can be used to add to an existing vector. For example, if an eighth male person were included in the data set, and his weight was 194 pounds, the existing vectors could be modified as follows.

```
> weight <- c(weight, 194)
> gender <- c(gender, "male")
> weight

[1] 123 157 205 199 223 140 105 194

> gender

[1] "female" "female" "male"    "female" "male"    "male"
[7] "female" "male"
```

#### 4.1.1 Types, conversion, coercion

Clearly it is important to distinguish between different types of vectors. For example, it makes sense to ask R to calculate the mean of the weights stored in `weight`, but does not make sense to ask R to compute the mean of the genders stored in `gender`. Vectors in R may have one of six different “types” including character, double, integer, logical, complex, and raw. Only the first four of these will be of interest below, and the distinction between double and integer will not be of great import. To illustrate logical vectors, imagine that each of the eight people in the data set was asked whether he or she was taking blood pressure medication, and the responses were coded as TRUE if the person answered yes, and FALSE if the person answered no.

```
> typeof(weight)
[1] "double"

> typeof(gender)
[1] "character"

> bp <- c(FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE)
> bp

[1] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE

> typeof(bp)
[1] "logical"
```

It may be surprising to see that the weight variable `weight` is of `double` type, even though its values all are integers. By default R creates a double type vector when numeric values are given via the `c()` function.

When it makes sense, it is possible to convert vectors to a different type. Consider the following examples.

```
> weight.int <- as.integer(weight)
> weight.int

[1] 123 157 205 199 223 140 105 194

> typeof(weight.int)

[1] "integer"

> weight.char <- as.character(weight)
> weight.char

[1] "123" "157" "205" "199" "223" "140" "105" "194"

> bp.double <- as.double(bp)
> bp.double

[1] 0 1 0 0 1 0 1 0

> gender.oops <- as.double(gender)

Warning:  NAs introduced by coercion

> gender.oops

[1] NA NA NA NA NA NA NA NA

> sum(bp)

[1] 3
```

The integer version of `weight` doesn't look any different, but it is stored differently, which can be important both for computational efficiency and for interfacing with other languages such as C++. As noted above, however, we will not worry about the distinction between integer and double types. Converting `weight` to character goes as expected: The character representations of the numbers replace the numbers themselves. Converting the logical vector `bp` to double is pretty straightforward too:

`FALSE` is converted to zero, and `TRUE` is converted to one. Now think about converting the character vector `gender` to a numeric double vector. It's not at all clear how to represent "female" and "male" as numbers. In fact in this case what R does is to create a character vector, but with each element set to `NA`, which is the representation of missing data.<sup>2</sup> Finally consider the code `sum(bp)`. Now `bp` is a logical vector, but when R sees that we are asking to sum this logical vector, it automatically converts it to a numerical vector and then adds the zeros and ones representing `FALSE` and `TRUE`.

R also has functions to test whether a vector is of a particular type.

```
> is.double(weight)
[1] TRUE

> is.character(weight)
[1] FALSE

> is.integer(weight.int)
[1] TRUE

> is.logical(bp)
[1] TRUE
```

## Coercion

Consider the following examples.

```
> xx <- c(1, 2, 3, TRUE)
> xx
[1] 1 2 3 1

> yy <- c(1, 2, 3, "dog")
> yy
[1] "1"   "2"   "3"   "dog"
```

---

<sup>2</sup>Missing data will be discussed in more detail below.

```
> zz <- c(TRUE, FALSE, "cat")
> zz

[1] "TRUE"  "FALSE" "cat"

> weight + bp

[1] 123 158 205 199 224 140 106 194
```

Vectors in R can only contain elements of one type. If more than one type is included in a `c()` function, R silently *coerces* the vector to be of one type. The examples illustrate the hierarchy—if any elements is a character, then the whole vector is character. If some elements are numeric (either integer or double) and other elements are logical, the whole vector is numeric. Note what happened when R was asked to add the numeric vector `weight` to the logical vector `bp`. The logical vector was silently coerced to be numeric, so that FALSE became zero and TRUE became one, and then the two numeric vectors were added.

### 4.1.2 Accessing specific elements of vectors

To access and possibly change specific elements of vectors, refer to the position of the element in square brackets. For example, `weight[4]` refers to the fourth element of the vector `weight`. Note that R starts the numbering of elements at 1, i.e., the first element of a vector `x` is `x[1]`.

```
> weight

[1] 123 157 205 199 223 140 105 194

> weight[5]

[1] 223

> weight[1:3]

[1] 123 157 205

> length(weight)

[1] 8

> weight[length(weight)]
```

```
[1] 194

> weight[]

[1] 123 157 205 199 223 140 105 194

> weight[3] <- 202
> weight

[1] 123 157 202 199 223 140 105 194
```

Note that including nothing in the square brackets results in the whole vector being returned.

Negative numbers in the square brackets tell R to omit the corresponding value. And a zero as a subscript returns nothing (more precisely, it returns a length zero vector of the appropriate type).

```
> weight[-3]

[1] 123 157 199 223 140 105 194

> weight[-length(weight)]

[1] 123 157 202 199 223 140 105

> lessWeight <- weight[-c(1, 3, 5)]
> lessWeight

[1] 157 199 140 105 194

> weight[0]

numeric(0)

> weight[c(0, 2, 1)]

[1] 157 123

> weight[c(-1, 2)]

Error in weight[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

Note that mixing zero and other nonzero subscripts is allowed, but mixing negative and positive subscripts is not allowed.

What about the (usual) case where we don't know the positions of the elements we want? For example possibly we want the weights of all females in the data. Later we will learn how to subset using logical indices, which is a very powerful way to access desired elements of a vector.

## 4.2 Factors

Categorical variables such as `gender` can be represented as character vectors. In many cases this simple representation is sufficient. Consider, however, two other categorical variables, one representing age via categories `youth`, `young adult`, `middle age`, `senior`, and another representing income via categories `lower`, `middle`, and `upper`. Suppose that for the small health data set, all the people are either middle aged or senior citizens. If we just represented the variable via a character vector, there would be no way to know that there are two other categories, representing youth and senior citizens, which happen not to be present in the data set. And for the income variable, the character vector representation does not explicitly indicate that there is an ordering of the levels.

Factors in R provide a more sophisticated way to represent categorical variables. Factors explicitly contain all possible levels, and allow ordering of levels.

```
> age <- c("middle age", "senior", "middle age", "senior",
+         "senior", "senior", "senior", "middle age")
> income <- c("lower", "lower", "upper", "middle", "upper",
+             "lower", "lower", "middle")
> age

[1] "middle age" "senior"      "middle age" "senior"
[5] "senior"     "senior"      "senior"     "middle age"

> income

[1] "lower"   "lower"   "upper"   "middle"  "upper"   "lower"
[7] "lower"   "middle"

> age <- factor(age, levels=c("youth", "young adult", "middle age",
+                             "senior"))
> age

[1] middle age senior      middle age senior      senior
[6] senior      senior      middle age
Levels: youth young adult middle age senior
```

```
> income <- factor(income, levels=c("lower", "middle", "upper"),
+                     ordered = TRUE)
> income

[1] lower lower upper middle upper lower lower middle
Levels: lower < middle < upper
```

In the factor version of `age` the levels are explicitly listed, so it is clear that the two included levels are not all the possible levels. And in the factor version of `income`, the ordering is explicit.

In many cases the character vector representation of a categorical variable is sufficient and easier to work with. In this book factors will not be used extensively. It is important to note that R often by default creates a factor when character data are read in, and sometimes it is necessary to use the argument `stringsAsFactors = FALSE` to explicitly tell R not to do this. This will be seen below when data frames are introduced.

## 4.3 Names of objects in R

There are few hard and fast restrictions on the names of objects (such as vectors) in R. But there are good practices, and things to avoid.

From the help page for `make.names` in R, the name of an R object is “syntactically valid” if the name “consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number” and is not one of the “reserved words” in R such as `if`, `TRUE`, `function`, etc. For example, `c45t.le_dog` and `.ty56` are both syntactically valid (although not very good names) while `4cats` and `log#@gopher` are not.

A few important comments about naming objects follow:

1. It is important to be aware that names of objects in R are case-sensitive, so `weight` and `Weight` do not refer to the same object.

```
> weight

[1] 123 157 202 199 223 140 105 194

> Weight

Error in eval(expr, envir, enclos): object 'Weight' not found
```

2. It is unwise to create an object with the same name as a built in R object such as the function `c` or the function `mean`. In earlier versions of R this could be

somewhat disastrous, but even in current versions, it is definitely not a good idea!

3. As much as possible, choose names that are informative. When creating a variable you may initially remember that `x` contains heights and `y` contains genders, but after a few hours, or a few days, or a few weeks, you probably will forget this. Better options are `Height` and `Gender`.
4. As much as possible, be consistent in how you name objects. In particular, decide how to separate multi-word names. Some options include:
  - Using case to separate: `BloodPressure` or `bloodPressure` for example
  - Using underscores to separate: `blood_pressure` for example
  - Using a period to separate: `blood.pressure` for example

## 4.4 Missing Data, Infinity, etc.

Most real-world data sets have variables where some observations are missing. In a longitudinal study participants may drop out. In a survey, participants may decide not to respond to certain questions. Statistical software should be able to represent missing data and to analyze data sets in which some data are missing.

In R, the value `NA` is used for a missing data value. Since missing values may occur in numeric, character, and other types of data, and since R requires that a vector contain only elements of one type, there are different types of `NA` values. Usually R determines the appropriate type of `NA` value automatically. It is worth noting that the default type for `NA` is logical, and that `NA` is NOT the same as the character string "`NA`".

```
> missingCharacter <- c("dog", "cat", NA, "pig", NA, "horse")
> missingCharacter

[1] "dog"    "cat"    NA       "pig"    NA       "horse"

> is.na(missingCharacter)

[1] FALSE FALSE  TRUE FALSE  TRUE FALSE

> missingCharacter <- c(missingCharacter, "NA")
> missingCharacter

[1] "dog"    "cat"    NA       "pig"    NA       "horse"  "NA"

> is.na(missingCharacter)
```

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

> allMissing <- c(NA, NA, NA)
> typeof(allMissing)

[1] "logical"
```

How should missing data be treated in computations, such as finding the mean or standard deviation of a variable? One possibility is to return `NA`. Another is to remove the missing value(s) and then perform the computation.

```
> mean(c(1, 2, 3, NA, 5))

[1] NA

> mean(c(1, 2, 3, NA, 5), na.rm = TRUE)

[1] 2.75
```

As this example shows, the default behavior for the `mean()` function is to return `NA`. If removal of the missing values and then computing the mean is desired, the argument `na.rm` is set to `TRUE`. Different R functions have different default behaviors, and there are other possible actions. Consulting the help for a function provides the details.

#### 4.4.1 Infinity and NaN

What happens if R code requests division by zero, or results in a number that is too large to be represented? Here are some examples.

```
> x <- 0:4
> x

[1] 0 1 2 3 4

> 1/x

[1]       Inf 1.0000000 0.5000000 0.3333333 0.2500000

> x/x

[1] NaN   1    1    1    1

> y <- c(10, 1000, 10000)
> 2^y

[1] 1.024000e+03 1.071509e+301           Inf
```

`Inf` and `-Inf` represent infinity and negative infinity (and numbers which are too large in magnitude to be represented as floating point numbers). `NaN` represents the result of a calculation where the result is undefined, such as dividing zero by zero. All of these are common to a variety of programming languages, including R.

## 4.5 Data Frames

Commonly, data is rectangular in form, with variables as columns and cases as rows. Continuing with the (contrived) data on weight, gender, and blood pressure medication, each of those variables would be a column of the data set, and each person's measurements would be a row. In R, such data are represented as a *data frame*.

```
> healthData <- data.frame(Weight = weight, Gender=gender, bp.meds = bp,
+                           stringsAsFactors=FALSE)
> healthData

  Weight Gender bp.meds
1     123 female FALSE
2     157 female TRUE
3     202 male   FALSE
4     199 female FALSE
5     223 male   TRUE
6     140 male   FALSE
7     105 female TRUE
8     194 male   FALSE

> names(healthData)

[1] "Weight"  "Gender"  "bp.meds"

> colnames(healthData)

[1] "Weight"  "Gender"  "bp.meds"

> names(healthData) <- c("Wt", "Gdr", "bp")
> healthData

  Wt    Gdr    bp
1 123 female FALSE
2 157 female TRUE
3 202 male   FALSE
4 199 female FALSE
5 223 male   TRUE
```

```

6 140 male FALSE
7 105 female TRUE
8 194 male FALSE

> rownames(healthData)

[1] "1" "2" "3" "4" "5" "6" "7" "8"

> names(healthData) <- c("Weight", "Gender", "bp.meds")

```

The `data.frame` function can be used to create a data frame (although it's more common to read a data frame into R from an external file, something that will be introduced later). The names of the variables in the data frame are given as arguments, as are the vectors of data that make up the variable's values. The argument `stringsAsFactors=FALSE` asks R not to convert character vectors into factors, which R does by default, to the dismay of many users. Names of the columns (variables) can be extracted and set via either `names` or `colnames`. In the example, the variable names are changed to `Wt`, `Gdr`, `bp` and then changed back to the original `Weight`, `Gender`, `bp.meds` in this way. Rows can be named also. In this case since specific row names were not provided, the default row names of "1", "2" etc. are used.

In the next example a built-in dataset called `mtcars` is made available by the `data` function, and then the first and last six rows are displayed using `head` and `tail`.

```

> data(mtcars)
> head(mtcars)

      mpg cyl disp  hp drat    wt  qsec vs am
Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1
Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1
Datsun 710    22.8   4 108  93 3.85 2.320 18.61  1  1
Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0
Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0
Valiant       18.1   6 225 105 2.76 3.460 20.22  1  0
                    gear carb
Mazda RX4        4     4
Mazda RX4 Wag    4     4
Datsun 710       4     1
Hornet 4 Drive   3     1
Hornet Sportabout 3     2
Valiant         3     1

> tail(mtcars)

```

|                | mpg  | cyl  | disp  | hp  | drat | wt    | qsec | vs | am |
|----------------|------|------|-------|-----|------|-------|------|----|----|
| Porsche 914-2  | 26.0 | 4    | 120.3 | 91  | 4.43 | 2.140 | 16.7 | 0  | 1  |
| Lotus Europa   | 30.4 | 4    | 95.1  | 113 | 3.77 | 1.513 | 16.9 | 1  | 1  |
| Ford Pantera L | 15.8 | 8    | 351.0 | 264 | 4.22 | 3.170 | 14.5 | 0  | 1  |
| Ferrari Dino   | 19.7 | 6    | 145.0 | 175 | 3.62 | 2.770 | 15.5 | 0  | 1  |
| Maserati Bora  | 15.0 | 8    | 301.0 | 335 | 3.54 | 3.570 | 14.6 | 0  | 1  |
| Volvo 142E     | 21.4 | 4    | 121.0 | 109 | 4.11 | 2.780 | 18.6 | 1  | 1  |
|                | gear | carb |       |     |      |       |      |    |    |
| Porsche 914-2  | 5    | 2    |       |     |      |       |      |    |    |
| Lotus Europa   | 5    | 2    |       |     |      |       |      |    |    |
| Ford Pantera L | 5    | 4    |       |     |      |       |      |    |    |
| Ferrari Dino   | 5    | 6    |       |     |      |       |      |    |    |
| Maserati Bora  | 5    | 8    |       |     |      |       |      |    |    |
| Volvo 142E     | 4    | 2    |       |     |      |       |      |    |    |

Note that the `mtcars` data frame does have non-default row names which give the make and model of the cars.

#### 4.5.1 Accessing specific elements of data frames

Data frames are two-dimensional, so to access a specific element (or elements) we need to specify both the row and column.

```
> mtcars[1, 4]

[1] 110

> mtcars[1:3, 3]

[1] 160 160 108

> mtcars[1:3, 2:3]

      cyl disp
Mazda RX4       6   160
Mazda RX4 Wag   6   160
Datsun 710     4   108

> mtcars[, 1]

[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8
[12] 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5
[23] 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

Note that `mtcars[,1]` returns ALL elements in the first column. This agrees with the behavior for vectors, where leaving a subscript out of the square brackets tells R to return all values. In this case we are telling R to return all rows, and the first column.

For a data frame there is another way to access specific columns, using the `$` notation.

```
> mtcars$mpg
[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8
[12] 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5
[23] 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4

> mtcars$cyl
[1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4
[29] 8 6 8 4

> mpg
Error in eval(expr, envir, enclos): object 'mpg' not found

> cyl
Error in eval(expr, envir, enclos): object 'cyl' not found

> weight
[1] 123 157 202 199 223 140 105 194
```

Notice that typing the variable name, such as `mpg`, without the name of the data frame (and a dollar sign) as a prefix, does not work. This is sensible. There may be several data frames that have variables named `mpg`, and just typing `mpg` doesn't provide enough information to know which is desired. But if there is a vector named `mpg` that is created outside a data frame, it will be retrieved when `mpg` is typed, which is why typing `weight` does work, since `weight` was created outside of a data frame, although ultimately it was incorporated into the `healthData` frame.

## 4.6 Lists

The third main data structure we will work with is a list. Technically a list is a vector, but one in which elements can be of different types. For example a list may have one element that is a vector, one element that is a data frame, and another element that is a function. Consider designing a function that fits a simple linear regression model

to two quantitative variables. We might want that function to compute and return several things such as

- The fitted slope and intercept (a numeric vector with two components)
- The residuals (a numeric vector with  $n$  components, where  $n$  is the number of data points)
- Fitted values for the data (a numeric vector with  $n$  components, where  $n$  is the number of data points)
- The names of the dependent and independent variables (a character vector with two components)

In fact R has a function, `lm`, which does this (and much more).

```
> mpgHpLinMod <- lm(mpg ~ hp, data = mtcars)
> mode(mpgHpLinMod)

[1] "list"

> names(mpgHpLinMod)

[1] "coefficients"   "residuals"      "effects"
[4] "rank"            "fitted.values"  "assign"
[7] "qr"              "df.residual"   "xlevels"
[10] "call"           "terms"         "model"

> mpgHpLinMod$coefficients

(Intercept)          hp
30.09886054 -0.06822828

> mpgHpLinMod$residuals

    Mazda RX4      Mazda RX4 Wag      Datsun 710
-1.59374995 -1.59374995 -0.95363068
Hornet 4 Drive   Hornet Sportabout   Valiant
-1.19374995  0.54108812 -4.83489134
Duster 360       Merc 240D        Merc 230
0.91706759   -1.46870730 -0.81717412
Merc 280         Merc 280C        Merc 450SE
-2.50678234   -3.90678234 -1.41777049
Merc 450SL       Merc 450SLC     Cadillac Fleetwood
-0.51777049   -2.61777049 -5.71206353
```

|                     |                   |               |
|---------------------|-------------------|---------------|
| Lincoln Continental | Chrysler Imperial | Fiat 128      |
| -5.02978075         | 0.29364342        | 6.80420581    |
| Honda Civic         | Toyota Corolla    | Toyota Corona |
| 3.84900992          | 8.23597754        | -1.98071757   |
| Dodge Challenger    | AMC Javelin       | Camaro Z28    |
| -4.36461883         | -4.66461883       | -0.08293241   |
| Pontiac Firebird    | Fiat X1-9         | Porsche 914-2 |
| 1.04108812          | 1.70420581        | 2.10991276    |
| Lotus Europa        | Ford Pantera L    | Ferrari Dino  |
| 8.01093488          | 3.71340487        | 1.54108812    |
| Maserati Bora       | Volvo 142E        |               |
| 7.75761261          | -1.26197823       |               |

The `lm` function returns a list (which in the code above has been assigned to the object `mpgHpLinMod`).<sup>3</sup> One component of the list is the length 2 vector of coefficients, while another component is the length 32 vector of residuals. The code also illustrates that named components of a list can be accessed using the dollar sign notation, as with data frames.

The `list` function is used to create lists.

```
> temporaryList <- list(first=weight, second=healthData,
+                         pickle=list(a = 1:10, b=healthData))
> temporaryList

$first
[1] 123 157 202 199 223 140 105 194

$second
  Weight Gender bp.meds
1    123 female FALSE
2    157 female TRUE
3    202   male FALSE
4    199 female FALSE
5    223   male TRUE
6    140   male FALSE
7    105 female TRUE
8    194   male FALSE

$pickle
$pickle$a
[1] 1 2 3 4 5 6 7 8 9 10
```

---

<sup>3</sup>The `mode` function returns the the type or storage mode of an object.

```
$pickle$b
  Weight Gender bp.meds
1     123 female   FALSE
2     157 female    TRUE
3     202 male    FALSE
4     199 female   FALSE
5     223 male    TRUE
6     140 male    FALSE
7     105 female   TRUE
8     194 male    FALSE
```

Here, for illustration, I assembled a list to hold some of the R data structures we have been working with in this chapter. The first list element, named `first`, holds the `weight` vector we created in Section 4.1, the second list element, named `second`, holds the `healthData` data frame, and the third list element, named `pickle`, holds a list with elements named `a` and `b` that hold a vector of values 1 through 10 and another copy of the `healthData` data frame, respectively. As this example shows, a list can contain another list.

### 4.6.1 Accessing specific elements of lists

We already have seen the dollar sign notation works for lists. In addition, the square bracket subsetting notation can be used. There is an added, somewhat subtle wrinkle—using either single or double square brackets.

```
> temporaryList$first
[1] 123 157 202 199 223 140 105 194

> mode(temporaryList$first)
[1] "numeric"

> temporaryList[[1]]
[1] 123 157 202 199 223 140 105 194

> mode(temporaryList[[1]])
[1] "numeric"

> temporaryList[1]
```

```
$first
[1] 123 157 202 199 223 140 105 194

> mode(temporaryList[1])

[1] "list"
```

Note the dollar sign and double bracket notation return a numeric vector, while the single bracket notation returns a list. Notice also the difference in results below.

```
> temporaryList[c(1, 2)]

$first
[1] 123 157 202 199 223 140 105 194

$second
  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE
3    202 male    FALSE
4    199 female   FALSE
5    223 male    TRUE
6    140 male    FALSE
7    105 female   TRUE
8    194 male    FALSE

> temporaryList[[c(1, 2)]]

[1] 157
```

The single bracket form returns the first and second elements of the list, while the double bracket form returns the second element in the first element of the list. Generally, do not put a vector of indices or names in a double bracket, you will likely get unexpected results. See, for example, the results below<sup>4</sup>.

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

Error in temporaryList[[c(1, 2, 3)]]: recursive indexing failed at level
2
```

---

<sup>4</sup>Try this example using only single brackets... it will return a list holding elements `first`, `second`, and `pickle`.

So, in summary, there are two main differences between using the single bracket [] and double bracket [[]]. First, the single bracket will return a list that holds the object(s) held at the given indices or names placed in the bracket, whereas the double brackets will return the actual object held at the index or name placed in the innermost bracket. Put differently, a single bracket can be used to access a range of list elements and will return a list, and a double bracket can only access a single element in the list and will return the object held at the index.

## 4.7 Subsetting with logical vectors

Consider the `healthData` data frame. How can we access only those weights which are more than 200? How can we access the genders of those whose weights are more than 200? How can we compute the mean weight of males and the mean weight of females? Or consider the `mtcars` data frame. How can we obtain the miles per gallon for all six cylinder cars? Both of these data sets are small enough that it would not be too onerous to extract the values by hand. But for larger or more complex data sets, this would be very difficult or impossible to do in a reasonable amount of time, and would likely result in errors.

R has a powerful method for solving these sorts of problems using a variant of the subsetting methods that we already have learned. When given a logical vector in square brackets, R will return the values corresponding to TRUE. To begin, focus on the `weight` and `gender` vectors created in Section 4.1.

The R code `weight > 200` returns a TRUE for each value of `weight` which is more than 200, and a FALSE for each value of `weight` which is less than or equal to 200. Similarly `gender == "female"` returns TRUE or FALSE depending on whether an element of `gender` is equal to `female`.

```
> weight
[1] 123 157 202 199 223 140 105 194

> weight > 200
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE

> gender[weight > 200]
[1] "male" "male"

> weight[weight > 200]
[1] 202 223
```

```
> gender == "female"
[1] TRUE TRUE FALSE TRUE FALSE FALSE TRUE FALSE
> weight[gender == "female"]
[1] 123 157 199 105
```

Consider the lines of R code one by one.

- `weight` instructs R to display the values in the vector `weight`.
- `weight > 200` instructs R to check whether each value in `weight` is greater than 200, and to return `TRUE` if so, and `FALSE` otherwise.
- The next line, `gender[weight > 200]`, does two things. First, inside the square brackets, it does the same thing as the second line, namely, returning `TRUE` or `FALSE` depending on whether a value of `weight` is or is not greater than 200. Second, each element of `gender` is matched with the corresponding `TRUE` or `FALSE` value, and is returned if and only if the corresponding value is `TRUE`. For example the first value of `gender` is `female`. Since the first `TRUE` or `FALSE` value is `FALSE`, the first value of `gender` is not returned. Only the third and fifth values of `gender`, both of which happen to be `male`, are returned. Briefly, this line returns the genders of those people whose weight is over 200 pounds.
- The fourth line of code, `weight[weight > 200]`, again begins by returning `TRUE` or `FALSE` depending on whether elements of `weight` are larger than 200. Then those elements of `weight` corresponding to `TRUE` values, are returned. So this line returns the weights of those people whose weights are more than 200 pounds.
- The fifth line returns `TRUE` or `FALSE` depending on whether elements of `gender` are equal to `female` or not.
- The sixth line returns the weights of those whose gender is `female`.

There are six comparison operators in R, `>`, `<`, `>=`, `<=`, `==`, `!=`. Note that to test for equality a “double equals sign” is used, while `!=` tests for inequality.

#### 4.7.1 Modifying or creating objects via subsetting

The results of subsetting can be assigned to a new (or existing) R object, and subsetting on the left side of an assignment is a common way to modify an existing R object.

```

> weight

[1] 123 157 202 199 223 140 105 194

> lightweight <- weight[weight < 200]
> lightweight

[1] 123 157 199 140 105 194

> x <- 1:10
> x

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

> x[x < 5] <- 0
> x

[1] 0 0 0 0 5 6 7 8 9 10

> y <- -3:9
> y

[1] -3 -2 -1 0 1 2 3 4 5 6 7 8 9

> y[y < 0] <- NA
> y

[1] NA NA NA 0 1 2 3 4 5 6 7 8 9

> rm(x)
> rm(y)

```

### 4.7.2 Logical subsetting and data frames

First consider the small and simple `healthData` data frame.

```

> healthData

  Weight Gender bp.meds
1    123 female   FALSE
2    157 female    TRUE

```

```

3    202 male FALSE
4    199 female FALSE
5    223 male TRUE
6    140 male FALSE
7    105 female TRUE
8    194 male FALSE

> healthData$Weight[healthData$Gender == "male"]

[1] 202 223 140 194

> healthData[healthData$Gender == "female", ]

  Weight Gender bp.meds
1    123 female FALSE
2    157 female TRUE
4    199 female FALSE
7    105 female TRUE

> healthData[healthData$Weight > 190, 2:3]

  Gender bp.meds
3 male FALSE
4 female FALSE
5 male TRUE
8 male FALSE

```

The first example is really just subsetting a vector, since the `$` notation creates vectors. The second two examples return subsets of the whole data frame. Note that the logical vector subsets the rows of the data frame, choosing those rows where the gender is female or the weight is more than 190. Note also that the specification for the columns (after the comma) is left blank in the first case, telling R to return all the columns. In the second case the second and third columns are requested explicitly.

Next consider the much larger and more complex `WorldBank` data frame. Recall, the `str` function displays the “structure” of an R object. Here is a look at the structure of several R objects.

```

> str(mtcars)

'data.frame': 32 obs. of 11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num 6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num 160 160 108 258 360 ...

```

```
$ hp : num 110 110 93 110 175 105 245 62 95 123 ...
$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt : num 2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num 16.5 17 18.6 19.4 17 ...
$ vs : num 0 0 1 1 0 1 0 1 1 1 ...
$ am : num 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num 4 4 1 1 2 1 4 2 2 4 ...
```

```
> str(temporaryList)
```

List of 3

```
$ first : num [1:8] 123 157 202 199 223 140 105 194
$ second:'data.frame': 8 obs. of 3 variables:
..$ Weight : num [1:8] 123 157 202 199 223 140 105 194
..$ Gender : chr [1:8] "female" "female" "male" "female" ...
..$ bp.meds: logi [1:8] FALSE TRUE FALSE FALSE TRUE FALSE ...
$ pickle:List of 2
..$ a: int [1:10] 1 2 3 4 5 6 7 8 9 10
..$ b:'data.frame': 8 obs. of 3 variables:
... . $ Weight : num [1:8] 123 157 202 199 223 140 105 194
... . $ Gender : chr [1:8] "female" "female" "male" "female" ...
... . $ bp.meds: logi [1:8] FALSE TRUE FALSE FALSE TRUE FALSE ...
```

```
> str(WorldBank)
```

```
'data.frame': 11880 obs. of 15 variables:
$ iso2c                      : chr "AD" "AD" "AD" "AD" ...
$ country                     : chr "Andorra" "Andorra" "Andorra" "Andorra" ...
$ year                         : int 1978 1979 1977 2007 1976 2011 2012 2008 1980 197...
$ fertility.rate               : num NA NA NA 1.18 NA NA NA 1.25 NA NA ...
$ life.expectancy              : num NA NA NA NA NA NA NA NA NA ...
$ population                   : num 33746 34819 32769 81292 31781 ...
$ GDP.per.capita.Current.USD : num 9128 11820 7751 39923 7152 ...
$ X15.to.25.yr.female.literacy: num NA NA NA NA NA NA NA NA NA ...
$ iso3c                        : chr "AND" "AND" "AND" "AND" ...
$ region                       : chr "Europe & Central Asia (all income levels)" "Eur...
$ capital                      : chr "Andorra la Vella" "Andorra la Vella" "Andorra la V...
$ longitude                     : num 1.52 1.52 1.52 1.52 1.52 ...
$ latitude                      : num 42.5 42.5 42.5 42.5 42.5 ...
$ income                        : chr "High income: nonOECD" "High income: nonOECD" "H...
$ lending                       : chr "Not classified" "Not classified" "Not classifie...
```

First we see that `mtcars` is a data frame which has 32 observations (rows) on each of 11 variables (columns). The names of the variables are given, along with their type (in this case, all numeric) and the first few values of each variable is given.

Second we see that `temporaryList` is a list with three components. Then each of the components is described separately, with the first few values again given.

Third we examine the structure of `WorldBank`. It is a data frame with 11880 observations on each of 15 variables. Some of these are character variables, some are numeric, and one, `year`, is integer. Looking at the first few values we see that some variables have missing values.

Consider creating a data frame which only has the observations from one year, say 1971. That's relatively easy. Just choose rows for which `year` is equal to 1971.

```
> WorldBank1971 <- WorldBank[WorldBank$year == 1971, ]
> dim(WorldBank1971)

[1] 216 15
```

The `dim` function returns the dimensions of a data frame, i.e., the number of rows and the number of columns. From `dim` we see that there are 216 cases from 1971.

Next, how can we create a data frame which only contains data from 1971, and also only contains cases for which there are no missing values in the fertility rate variable? R has a built in function `is.na` which returns `TRUE` if the observation is missing and returns `FALSE` otherwise. And `!is.na` returns the negation, i.e., it returns `FALSE` if the observation is missing and `TRUE` if the observation is not missing.

```
> WorldBank1971$fertility.rate[1:25]

[1]      NA 6.512000 7.671000 3.517000 4.933000 3.118000
[7] 7.264000 3.104000      NA 2.200000 2.961000 2.788000
[13] 4.479000 2.260301 2.775000 2.949000 6.942000 2.210000
[19] 6.657000 2.100000 6.293000 7.329000 6.786000      NA
[25] 5.771000

> !is.na(WorldBank1971$fertility.rate[1:25])

[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[10] TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[19] TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE

> WorldBank1971 <- WorldBank1971[!is.na(WorldBank1971$fertility.rate),
+      ]
> dim(WorldBank1971)

[1] 193 15
```

From `dim` we see that there are 193 cases from 1971 with non-missing fertility rate data.

Return attention now to the original `WorldBank` data frame with data not only from 1971. How can we extract only those cases (rows) which have NO missing data? Consider the following simple example:

```
> temporaryDataFrame <- data.frame(V1 = c(1, 2, 3, 4, NA),
+                                     V2 = c(NA, 1, 4, 5, NA),
+                                     V3 = c(1, 2, 3, 5, 7))
> temporaryDataFrame

  V1 V2 V3
1  1 NA  1
2  2  1  2
3  3  4  3
4  4  5  5
5 NA NA  7

> is.na(temporaryDataFrame)

      V1     V2     V3
[1,] FALSE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE FALSE FALSE
[5,]  TRUE  TRUE FALSE

> rowSums(is.na(temporaryDataFrame))

[1] 1 0 0 0 2
```

First notice that `is.na` will test each element of a data frame for missingness. Also recall that if R is asked to sum a logical vector, it will first convert the logical vector to numeric and then compute the sum, which effectively counts the number of elements in the logical vector which are TRUE. The `rowSums` function computes the sum of each row. So `rowSums(is.na(temporaryDataFrame))` returns a vector with as many elements as there are rows in the data frame. If an element is zero, the corresponding row has no missing values. If an element is greater than zero, the value is the number of variables which are missing in that row. This gives a simple method to return all the cases which have no missing data.

```
> dim(WorldBank)
```

```
[1] 11880    15
```

```
> WorldBankComplete <- WorldBank[rowSums(is.na(WorldBank)) == 0,
+      ]
> dim(WorldBankComplete)

[1] 564 15
```

Out of the 11880 rows in the original data frame, only 564 have no missing observations!

## 4.8 Patterned data

Sometimes it is useful to generate all the integers from 1 through 20, to generate a sequence of 100 points equally spaced between 0 and 1, etc. The R functions `seq()` and `rep()` as well as the “colon operator” `:` help to generate such sequences.

The colon operator generates a sequence of values with increments of 1 or  $-1$ .

```
> 1:10

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

> -5:3

[1] -5 -4 -3 -2 -1 0 1 2 3

> 10:4

[1] 10 9 8 7 6 5 4

> pi:7

[1] 3.141593 4.141593 5.141593 6.141593
```

The `seq()` function generates either a sequence of pre-specified length or a sequence with pre-specified increments.

```
> seq(from = 0, to = 1, length = 11)

[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(from = 1, to = 5, by = 1/3)
```

```
[1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667
[7] 3.000000 3.333333 3.666667 4.000000 4.333333 4.666667
[13] 5.000000

> seq(from = 3, to = -1, length = 10)

[1] 3.0000000 2.5555556 2.1111111 1.6666667 1.2222222
[6] 0.7777778 0.3333333 -0.1111111 -0.5555556 -1.0000000
```

The `rep()` function replicates the values in a given vector.

```
> rep(c(1, 2, 4), length = 9)

[1] 1 2 4 1 2 4 1 2 4

> rep(c(1, 2, 4), times = 3)

[1] 1 2 4 1 2 4 1 2 4

> rep(c("a", "b", "c"), times = c(3, 2, 7))

[1] "a" "a" "a" "b" "b" "c" "c" "c" "c" "c" "c" "c"
```

### Exercise 3

Learning objectives: create, subset, and manipulate vector contents and attributes; summarize vector data using R `table` and other functions; generate basic graphics using vector data.

### Exercise 4

Learning objectives: use functions to describe data frame characteristics; summarize and generate basic graphics for variables held in data frames; apply the subset function with logical operators; illustrate `NA`, `NaN`, `Inf`, and other special values occur; recognize the implications of using floating point arithmetic with logical operators.

**Exercise 5**

Learning objectives: practice with lists, data frames, and associated functions; summarize variables held in lists and data frames; work with R's linear regression `lm` function output; review logical subsetting of vectors for partitioning and assigning of new values; generate and visualize data from mathematical functions.

# Chapter 5

## Graphics in R

R can be used to create a vast array of graphical representations of data. Creating “standard” graphical displays is straightforward, but a main strength of R is the ability to customize graphical displays to create either non-standard graphics or to modify more standard graphical displays to create publication-ready versions.

There are several packages available in R for creating graphics. The two leading packages are the `graphics` package, which comes with your base install of R, and the `ggplot2` package, which must be installed and made available by the user.<sup>1</sup> Knowing how to use both the `graphics` and `ggplot2` packages is worthwhile. For beginners `ggplot2` has somewhat simpler syntax, and also produces excellent graphics without much tinkering, so the focus in this book will be on `ggplot2`.

The `gg` in `ggplot2` stands for *Grammar of Graphics*. The package provides a unified and logical way to describe graphical displays, whether scatter plots, histograms, bar charts, or some other type of graphic. The grammar describes the mapping from data to the graphical display’s aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars). As will become obvious, once this grammar is mastered for a particular type of plot such as a scatter plot, it is easy to transfer this knowledge to other types of graphics.

Once you work through this chapter, the best place to learn more about `ggplot2` is from the package’s official book [Wickham and Sievert \[2016\]](#) by Hadley Wickham. It is available on-line in digital format from MSU’s library. The book goes into much more depth on the theory underlying the grammar and syntax, and has many examples on solve practical graphical problems. In addition to the free on-line version available through MSU, the book’s source code is available at <https://github.com/hadley/ggplot2-book>.

Another useful resource is the `ggplot2` extensions guide <http://www.ggplot2-exts.org>. This site lists packages that extend `ggplot2`. It’s a good place to start if you’re trying to do something that seems hard with `ggplot2`. We’ll explore a few of these extension packages toward the end of this chapter.

---

<sup>1</sup>Other graphics packages include `lattice` and `grid`

## 5.1 Scatter Plots

Scatter plots are a workhorse of data visualization and provide a good entry point to the `ggplot2` system. Begin by considering a simple and classic data set sometimes called *Fisher's Iris Data*. These data are available in R.

```
> data(iris)
> str(iris)

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

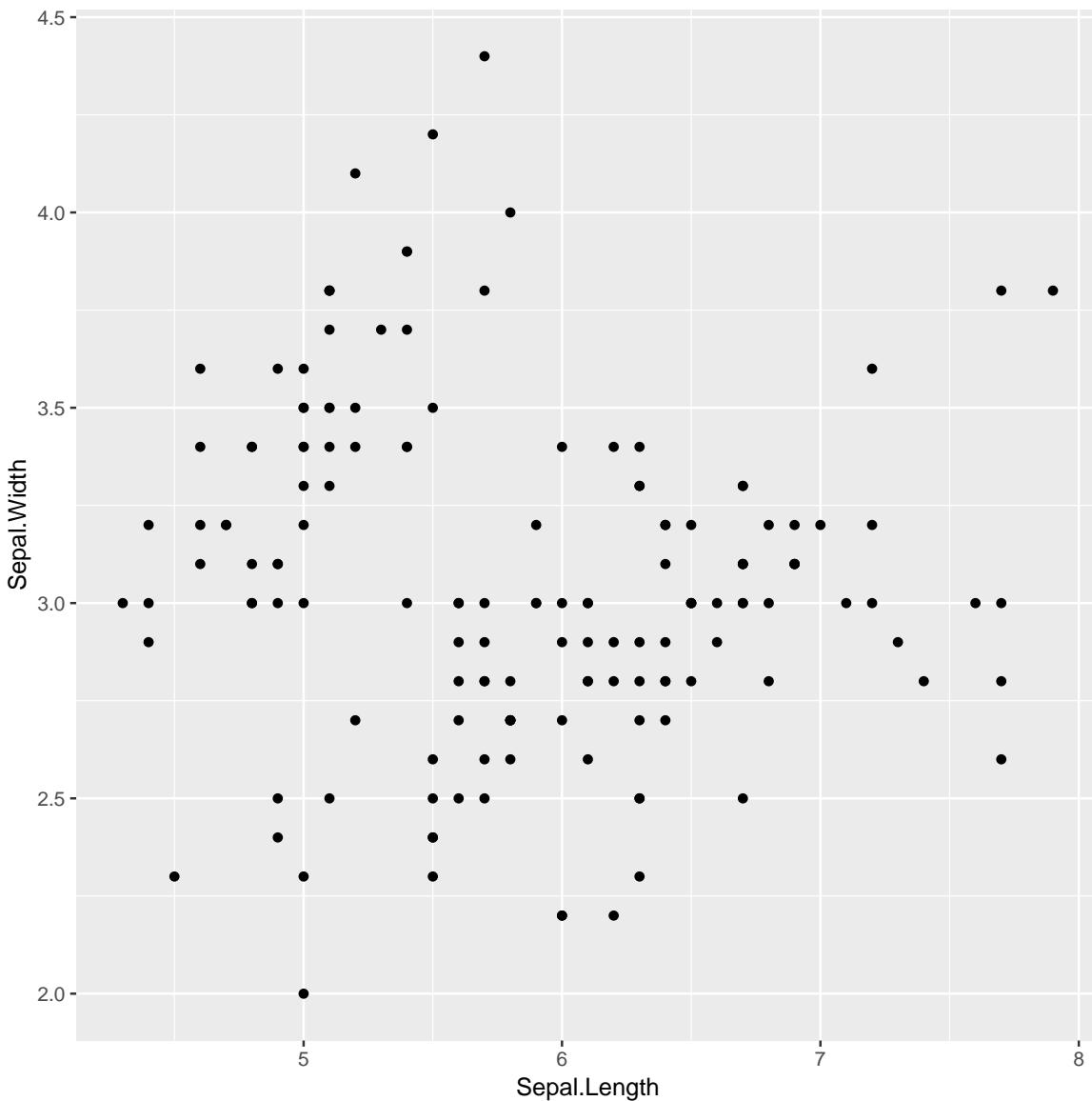
The data contain measurements on petal and sepal length and width for 150 iris plants. The plants are from one of three species, and the species information is also included in the data frame. The data are commonly used to test classification methods, where the goal would be to correctly determine the species based on the four length and width measurements. To get a preliminary sense of how this might work, we can draw some scatter plots of length versus width. Recall that `ggplot2` is not available by default, so we first have to download and install the package.

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

Once this is done the package is installed on the local hard drive, and we can use the `library` function to make the package available during the current R session.

Next a basic scatter plot is drawn. We'll keep the focus on sepal length and width, but of course similar plots could be drawn using petal length and width. The prompt is not displayed below, since the continuation prompt `+` can cause confusion.

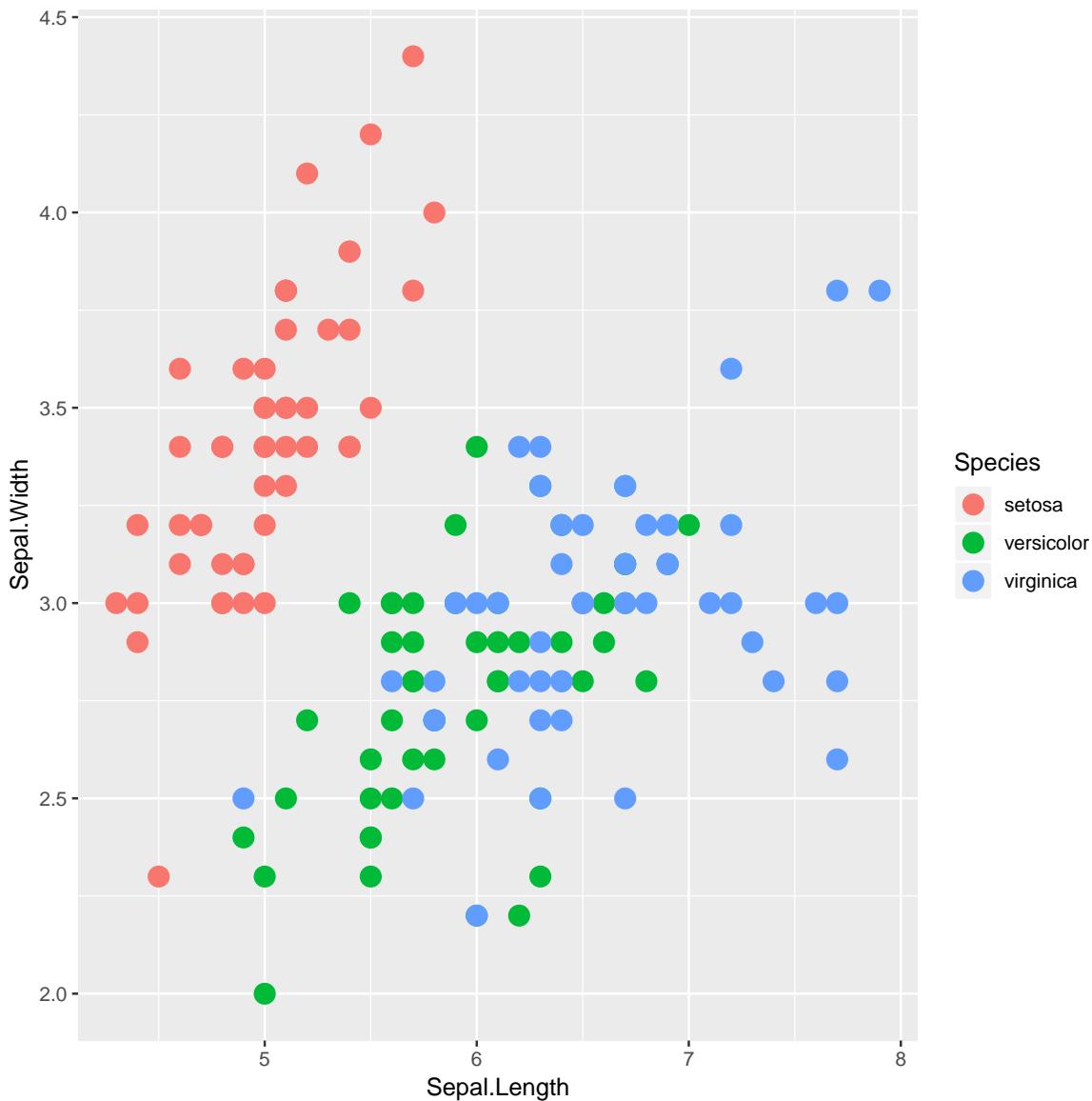
```
library(ggplot2)
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()
```



In this case the first argument to the `ggplot` function is the name of the data frame. Second, the `aes` (short for aesthetics) function specifies the mapping to the `x` and `y` axes. By itself the `ggplot` function as written doesn't tell R what sort of graphical display is desired. That is done by adding a `geom` (short for geometry) specification, in this case `geom_point`.

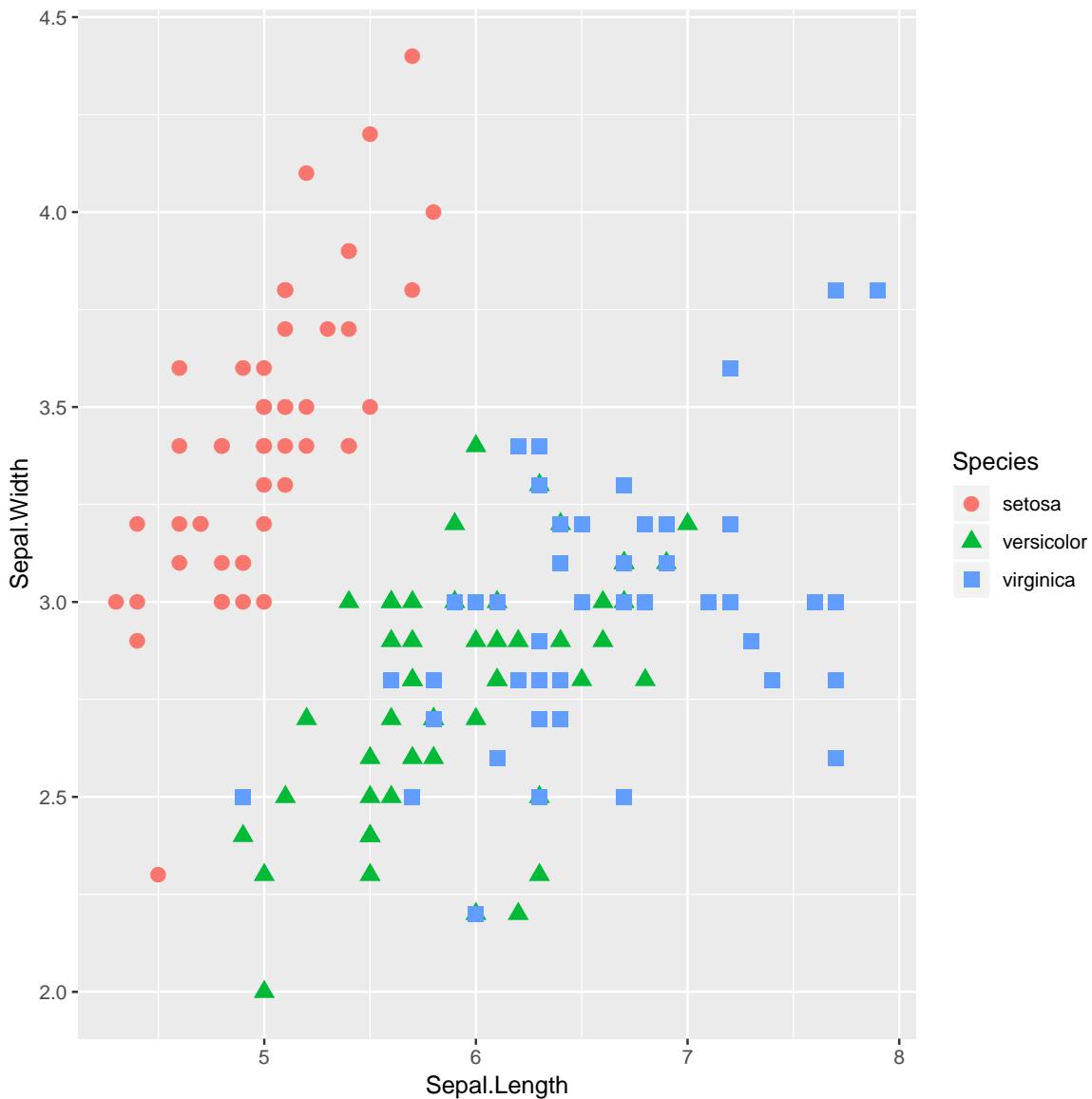
Looking at the scatter plot and thinking about the focus of finding a method to classify the species, two thoughts come to mind. First, the plot might be improved by increasing the size of the points. And second, using different colors for the points corresponding to the three species would help.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size = 4, aes(color=Species))
```



Notice that a legend showing what the colors represent is automatically generated and included in the graphic. Next, the size of the points seems a bit big now, and it might be helpful to use different shapes for the different species.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size = 3, aes(color=Species, shape=Species))
```



Here we see that the legend automatically changes to include species specific color and shape. The size of the points seems more appropriate.

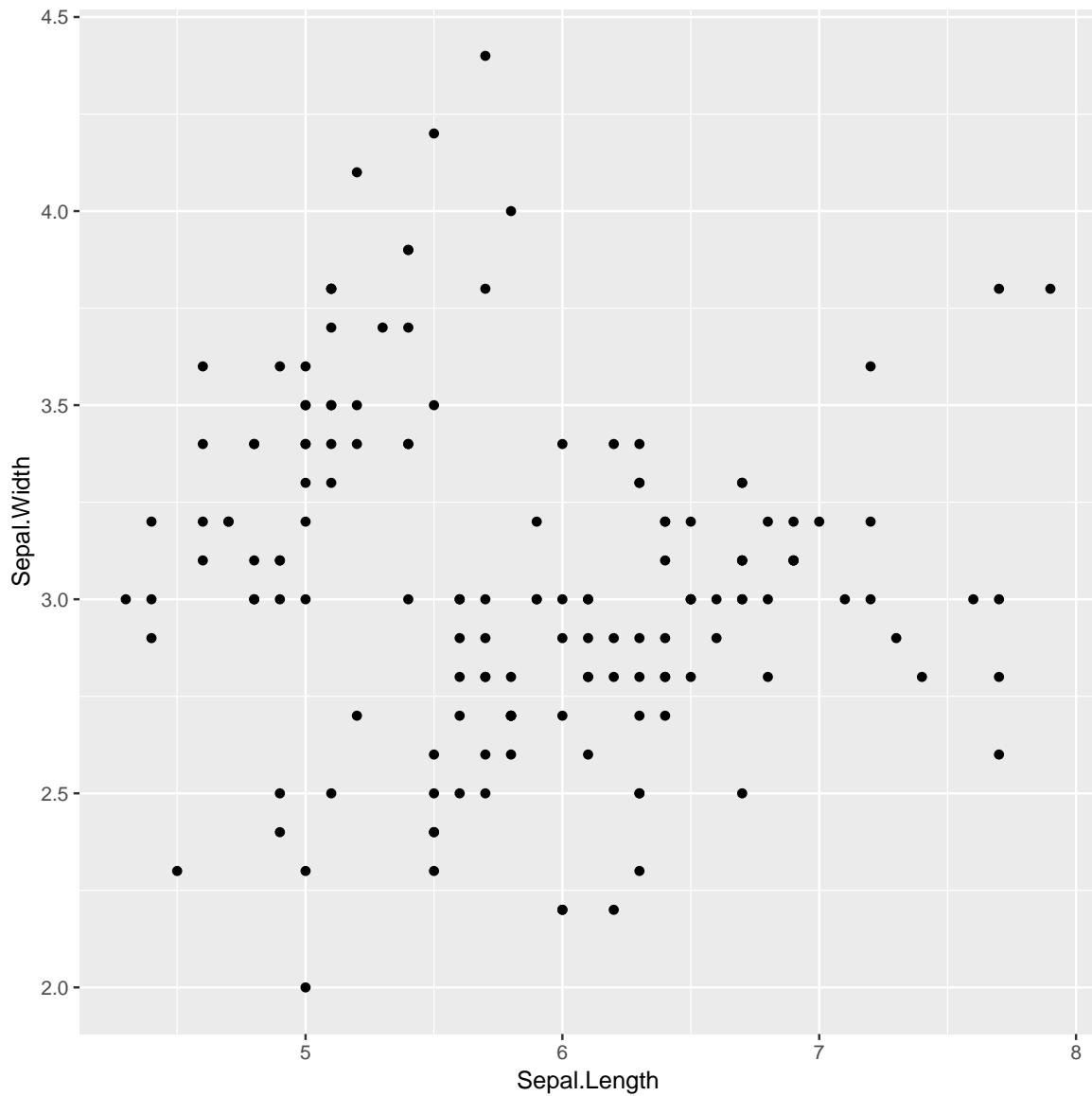
### 5.1.1 Structure of a typical ggplot

The examples above start with the function `ggplot()`, which takes as arguments the data frame containing the data to be plotted as well as a mapping from the data to the axes, enclosed by the `aes()` function. Next a `geom` function, in the above case `geom_point()`, is added. It might just specify the geometry, but also might specify aspects such as size, color, or shape.

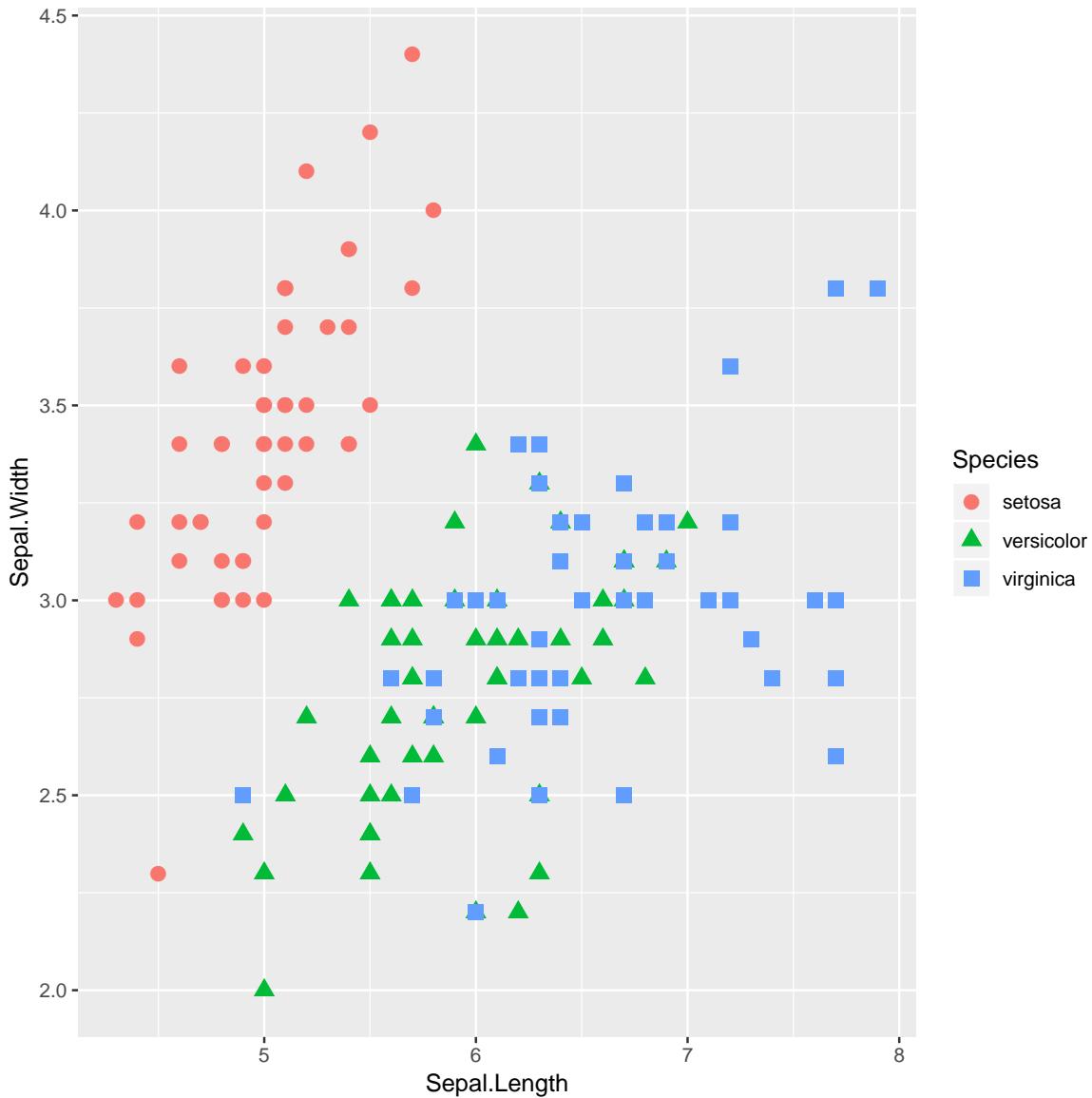
Typically many graphics are created and discarded in the search for an informative graphic, and often the initial specification of data and basic aesthetics from `ggplot()` stays the same in all the attempts. In such a case it can be helpful to assign that

portion of the graphic to an R object, both to minimize the amount of typing and to keep certain aspects of all the graphics constant. Here's how that could be done for the graphics above.

```
iris.p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width))  
iris.p + geom_point()
```



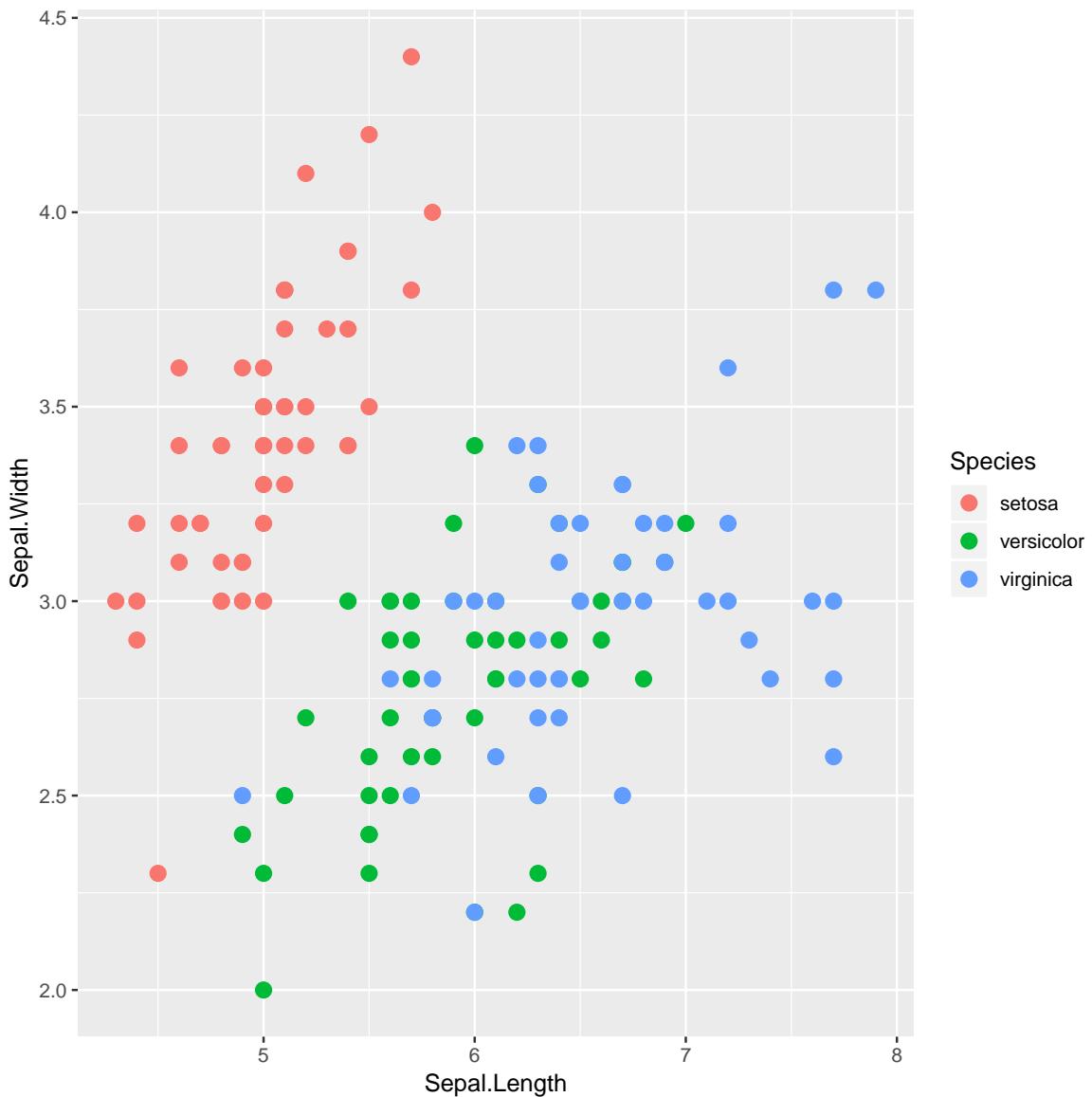
```
iris.p + geom_point(size = 3, aes(color = Species, shape = Species))
```



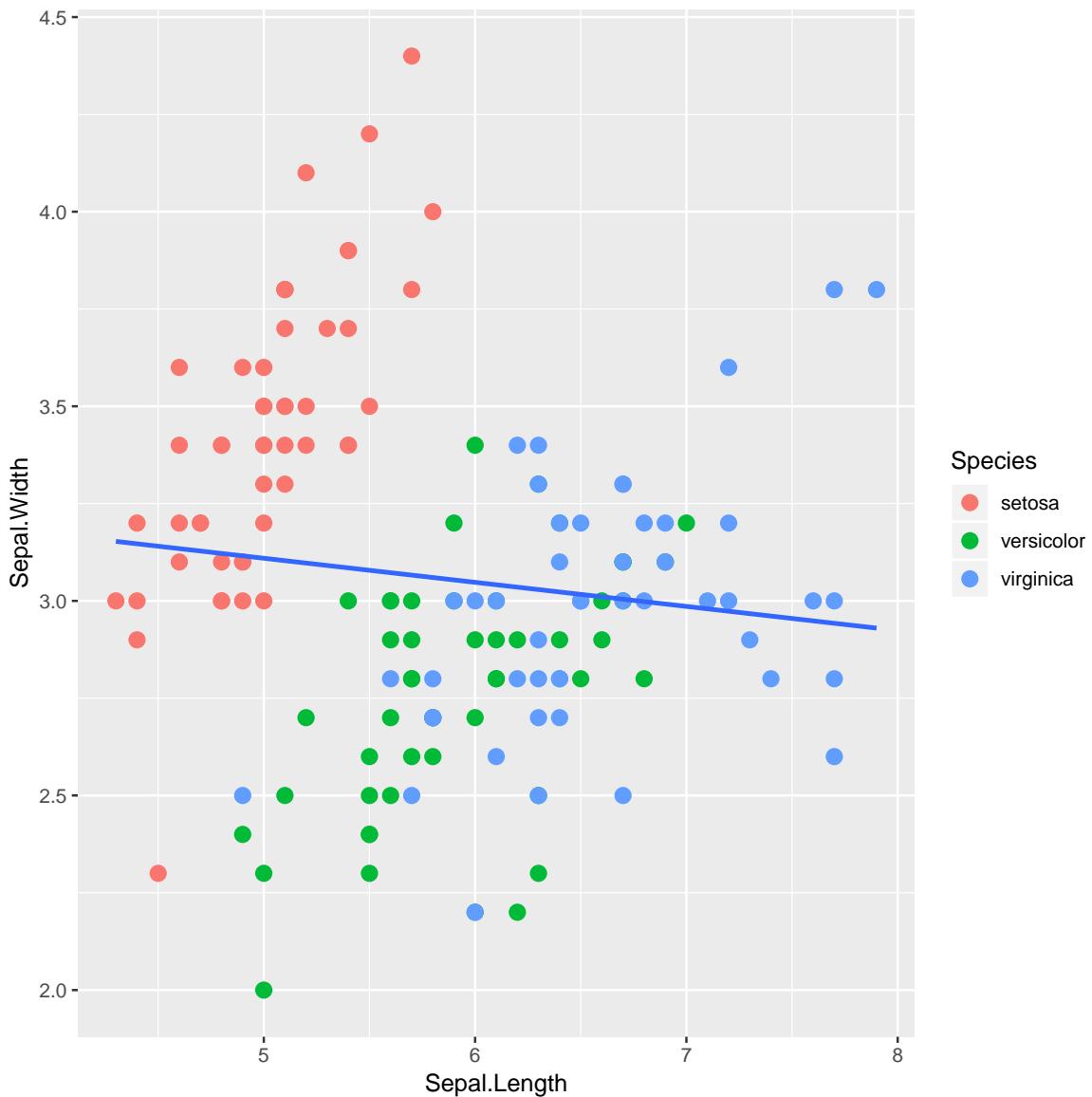
### 5.1.2 Adding lines to a scatter plot

To add a fitted least squares line to a scatter plot, use `stat_smooth`, which adds a smoother (possibly a least squares line, possibly a smooth curve fit to the data, etc.). The argument `method = lm` specifies a line fitted by least squares, and the argument `se = FALSE` suppresses the default display of a confidence band around the line or curve which was fit to the data.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size=3, aes(color=Species))
```

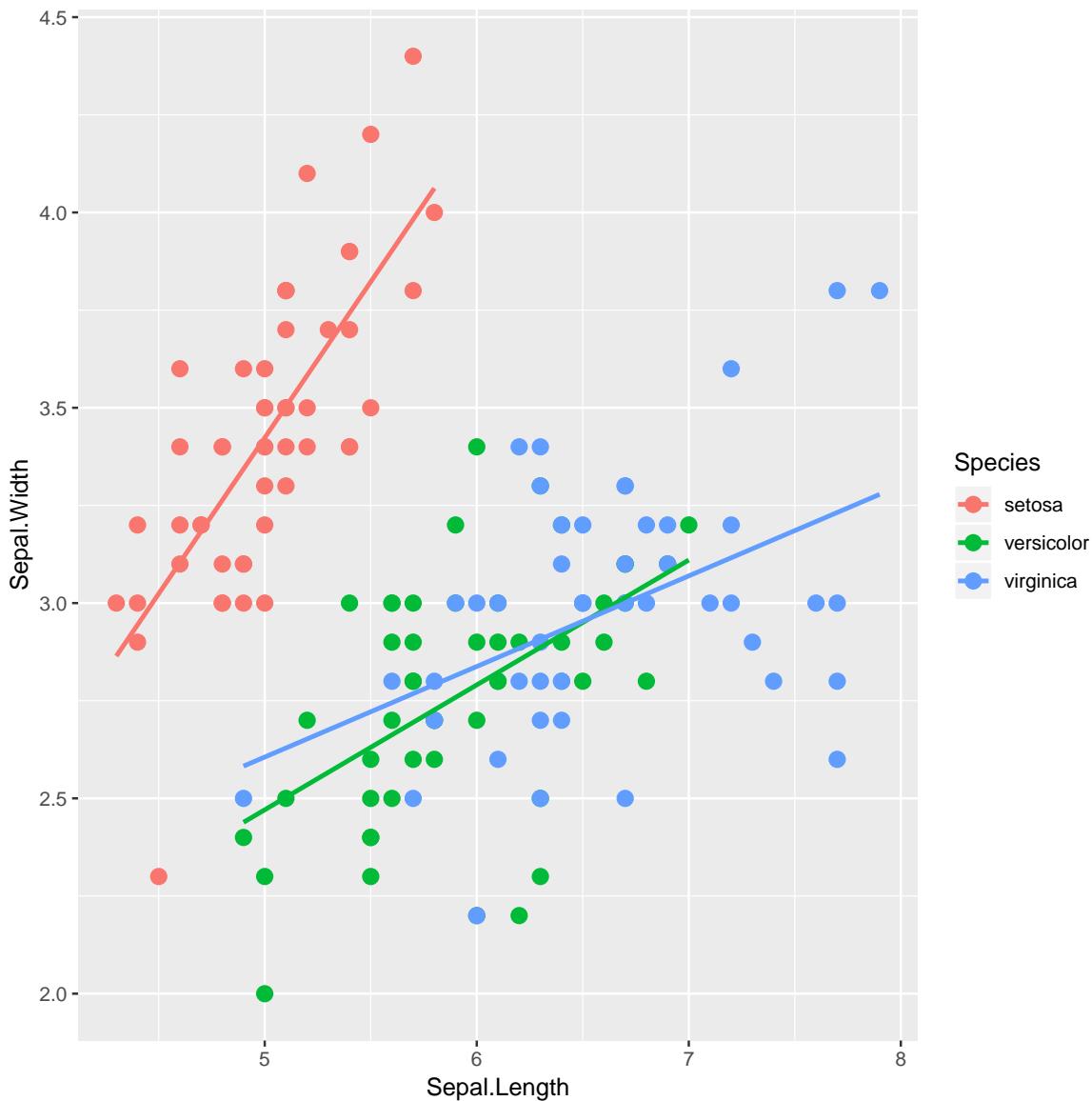


```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size=3, aes(color=Species)) +
  stat_smooth(method = lm, se=FALSE)
```



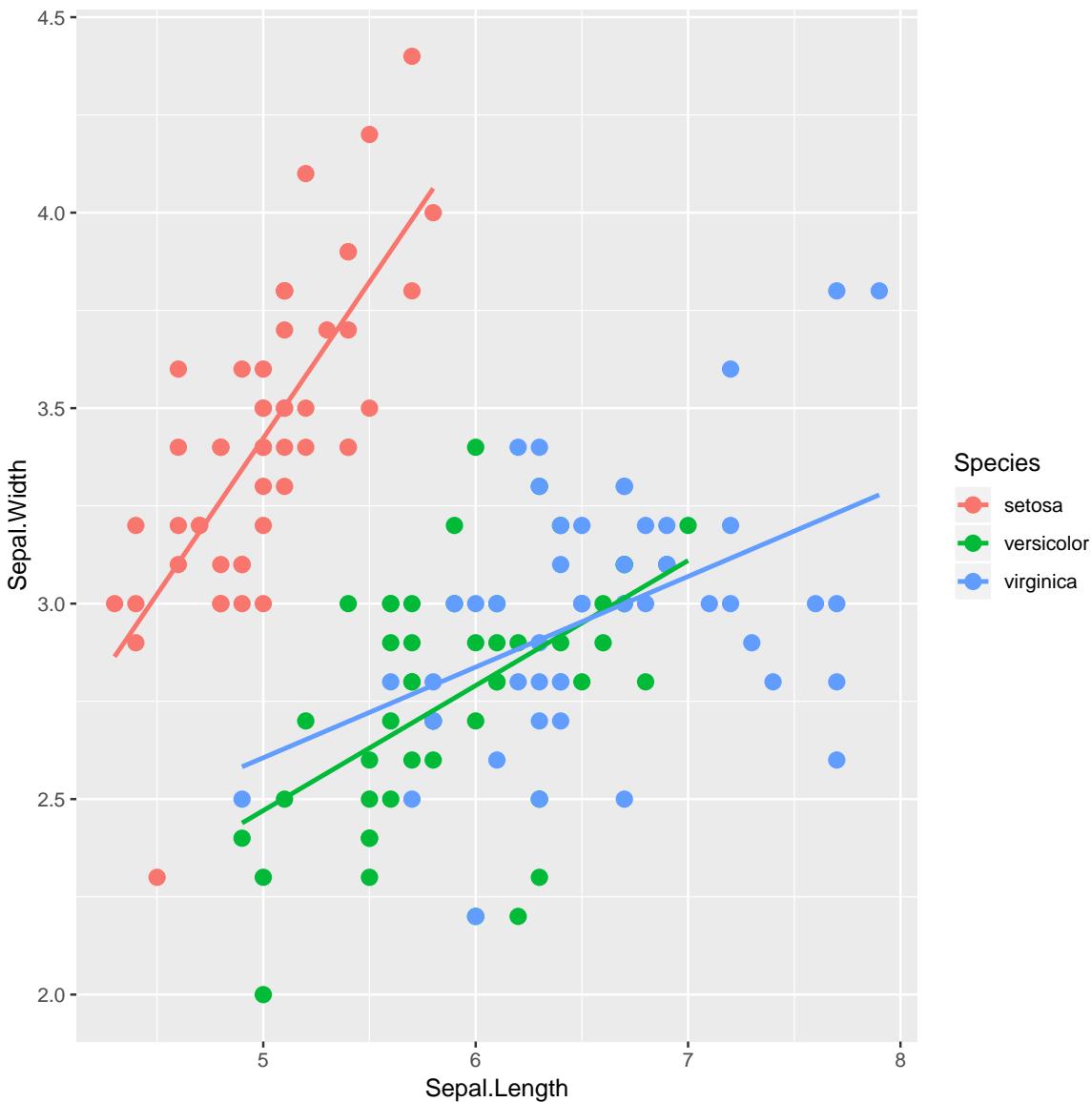
For the iris data, it probably makes more sense to fit separate lines by species. This can be specified using the `aes()` function inside `stat_smooth()`.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size=3, aes(color=Species)) +
  stat_smooth(method = lm, se=FALSE, aes(color=Species))
```



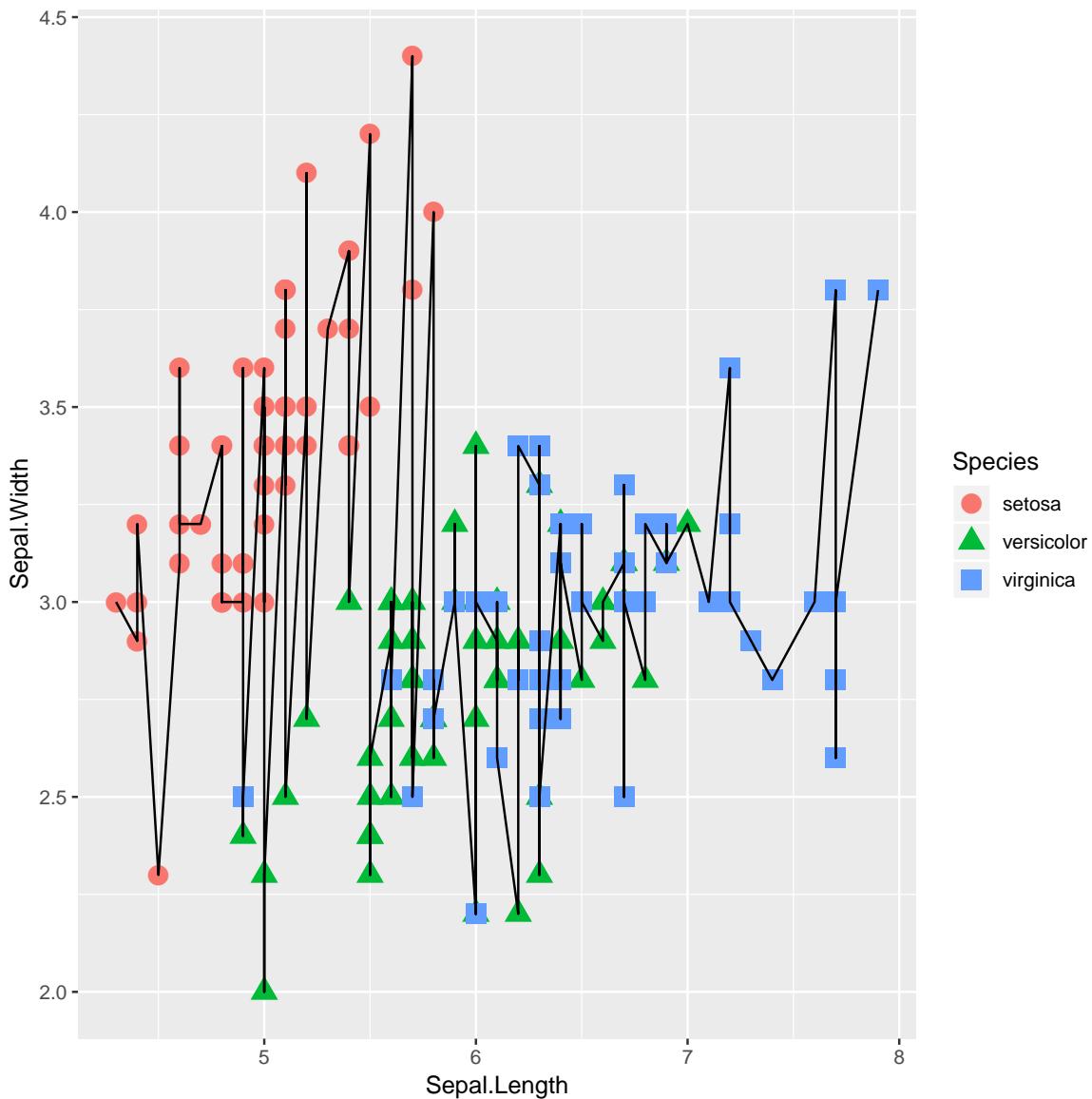
In this case we specified the same color aesthetic for the points and the lines. If we know we want this color aesthetic (colors corresponding to species) for all aspects of the graphic, we can specify it in the main `ggplot()` function:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point(size=3) + stat_smooth(method = lm, se=FALSE)
```

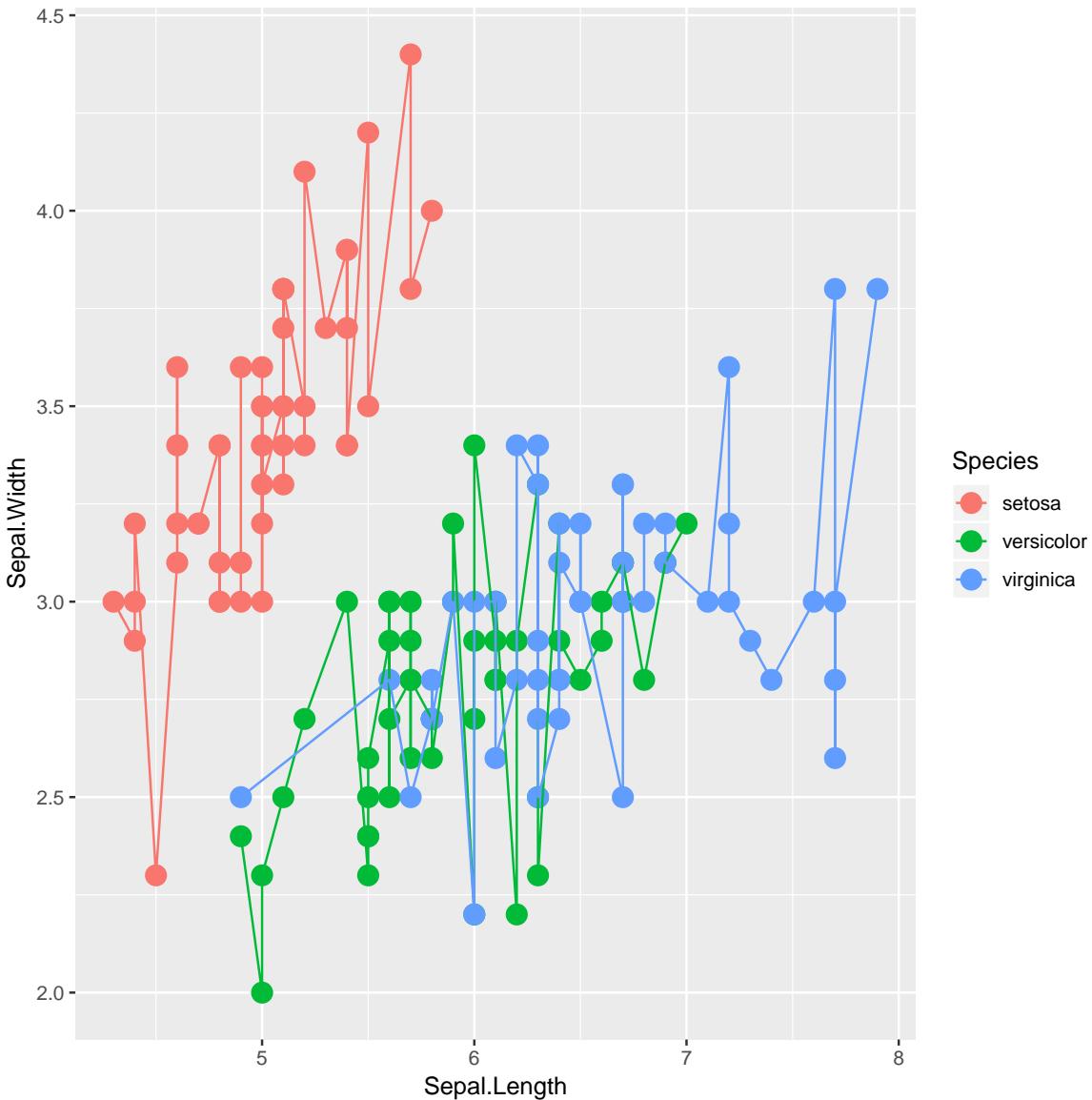


Another common use of line segments in a graphic is to connect the points in order, accomplished via the `geom_line()` function. Although it is not clear why this helps in understanding the iris data, the technique is illustrated next, first doing this for all the points in the graphic, and second doing this separately for the three species.

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(size = 4, aes(color=Species, shape = Species)) +
  geom_line()
```



```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point(size = 4, aes(color=Species)) +  
  geom_line(aes(color=Species))
```



## 5.2 Labels, axes, text, etc.

The default settings of `ggplot2` often produce excellent graphics, but once a graphic is chosen for dissemination, the user will likely want to customize things like the title, axes, etc. In this section some tools for customization are presented. Most will be illustrated in the context of a data set on crime rates in the 50 states in the United States. These data were made available by Nathan Yau at <http://flowingdata.com/2010/11/23/how-to-make-bubble-charts/>. The data include crime rates per 100,000 people for various crimes such as murder and robbery, and also include each state's population. The crime rates are from the year 2005, while the population numbers are from the year 2008, but the difference in population between the years is not great,

and the exact population is not particularly important for what we'll do below.

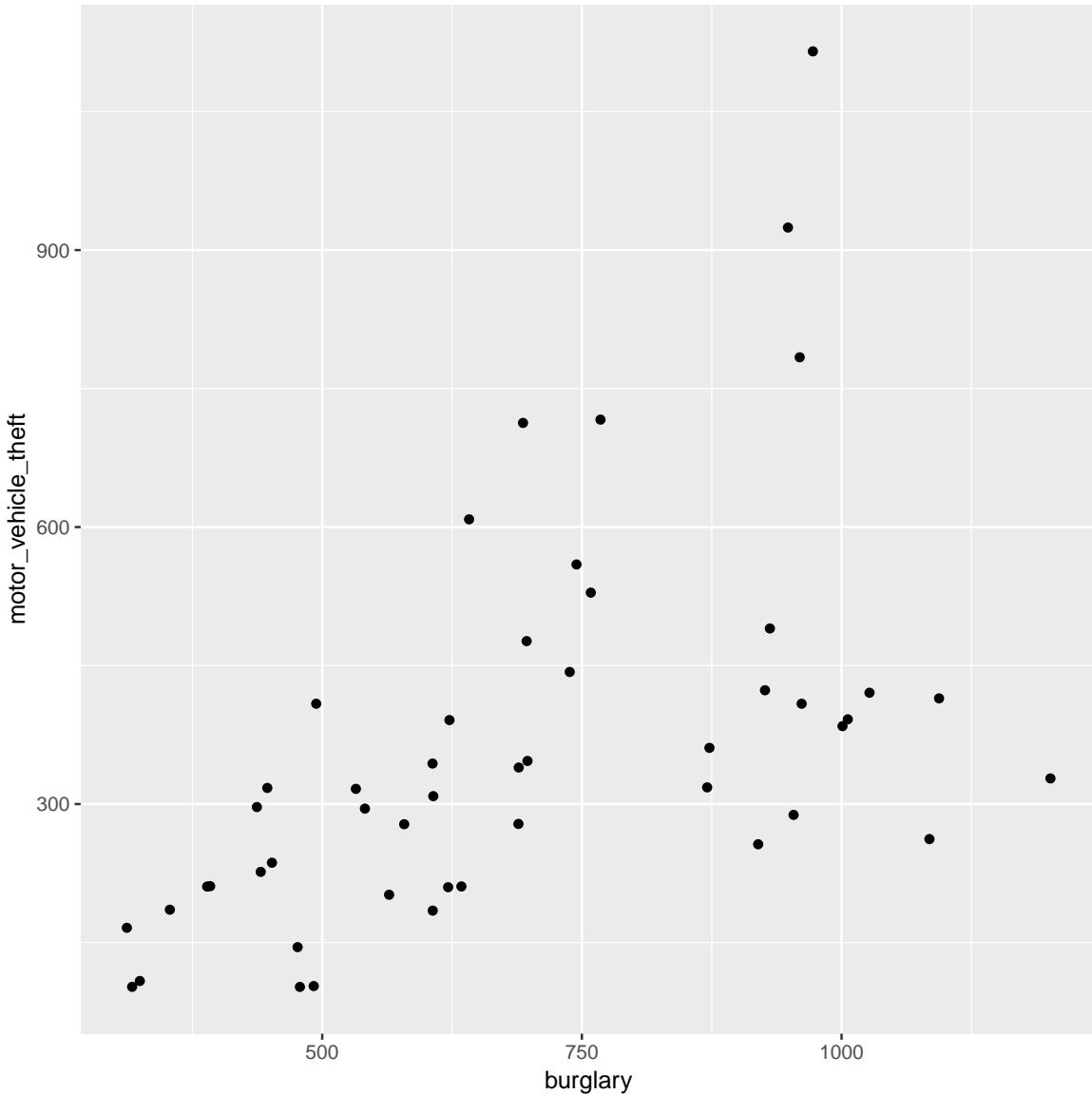
First, read in the data, examine its structure, and produce a simple scatter plot of motor vehicle theft versus burglary.

```
> u.crime <- "http://blue.for.msu.edu/FOR875/data/crimeRatesByState2005.csv"
> crime <- read.csv(u.crime, header=TRUE)
> str(crime)

'data.frame': 50 obs. of 9 variables:
 $ state           : Factor w/ 50 levels "Alabama","Alaska",...: 1 2 3 4 5 6 7 8 ...
 $ murder          : num  8.2 4.8 7.5 6.7 6.9 3.7 2.9 4.4 5 6.2 ...
 $ Forcible_rate   : num  34.3 81.1 33.8 42.9 26 43.4 20 44.7 37.1 23.6 ...
 $ Robbery         : num  141.4 80.9 144.4 91.1 176.1 ...
 $ aggravated_assult: num  248 465 327 387 317 ...
 $ burglary        : num  954 622 948 1085 693 ...
 $ larceny_theft   : num  2650 2599 2965 2711 1916 ...
 $ motor_vehicle_theft: num  288 391 924 262 713 ...
 $ population      : int  4627851 686293 6500180 2855390 36756666 4861515 3501252 8
```

```
> ggplot(data <- crime, aes(x = burglary, y = motor_vehicle_theft)) +
+     geom_point()
```



### 5.2.1 Labels

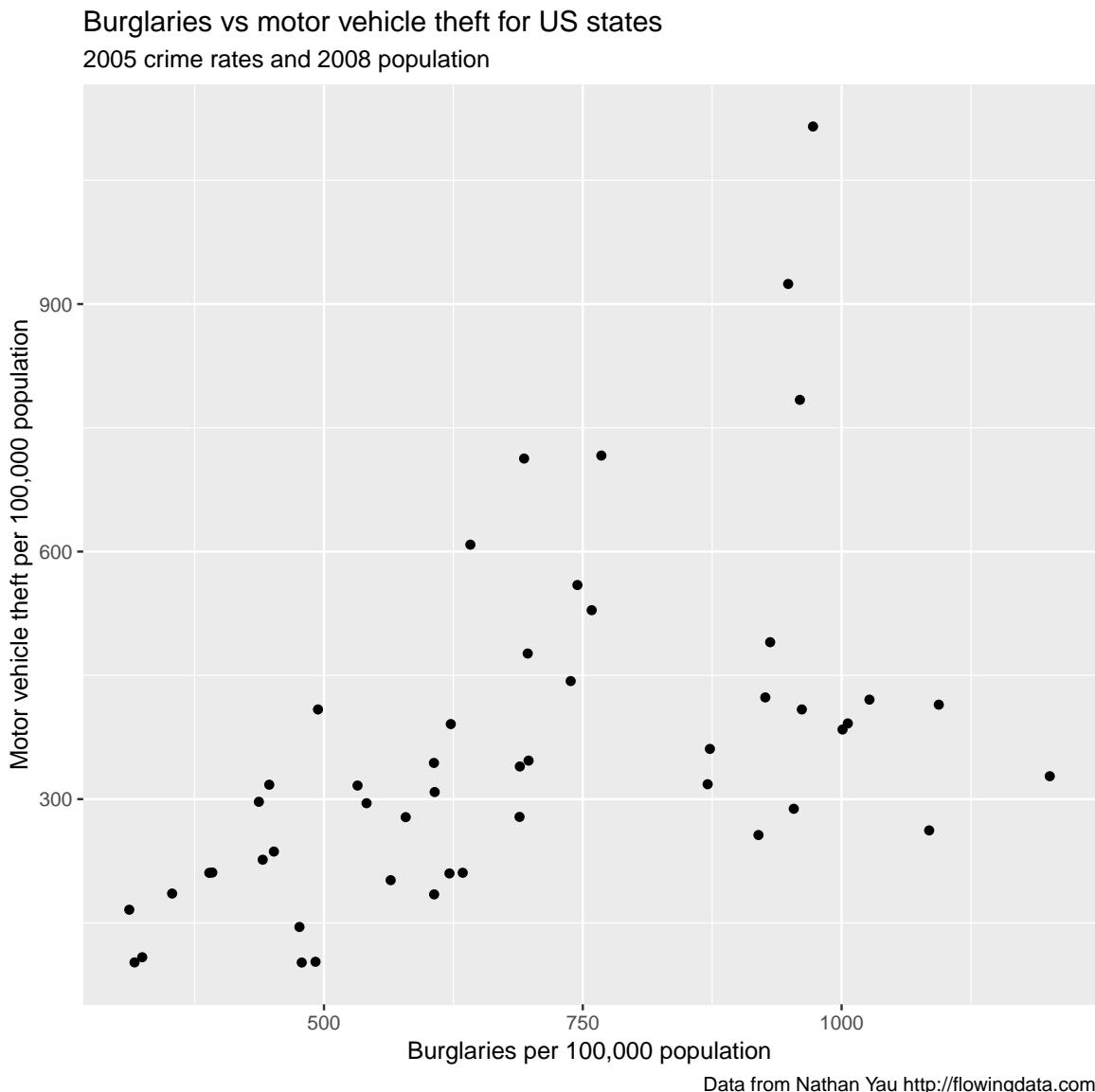
By default axis and legend labels are the names of the relevant columns in the data frame. While convenient, we often want to customize these labels. Here we use `labs()` to change the x and y axis labels and other descriptive text.<sup>2</sup>

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft)) +
  geom_point() +
  labs(x = "Burglaries per 100,000 population",
       y = "Motor vehicle theft per 100,000 population",
```

---

<sup>2</sup>Axis and legend labels can also be set in the individual scales, see the subsequent sections.

```
title = "Burglaries vs motor vehicle theft for US states",
subtitle = "2005 crime rates and 2008 population",
caption = "Data from Nathan Yau http://flowingdata.com"
)
```

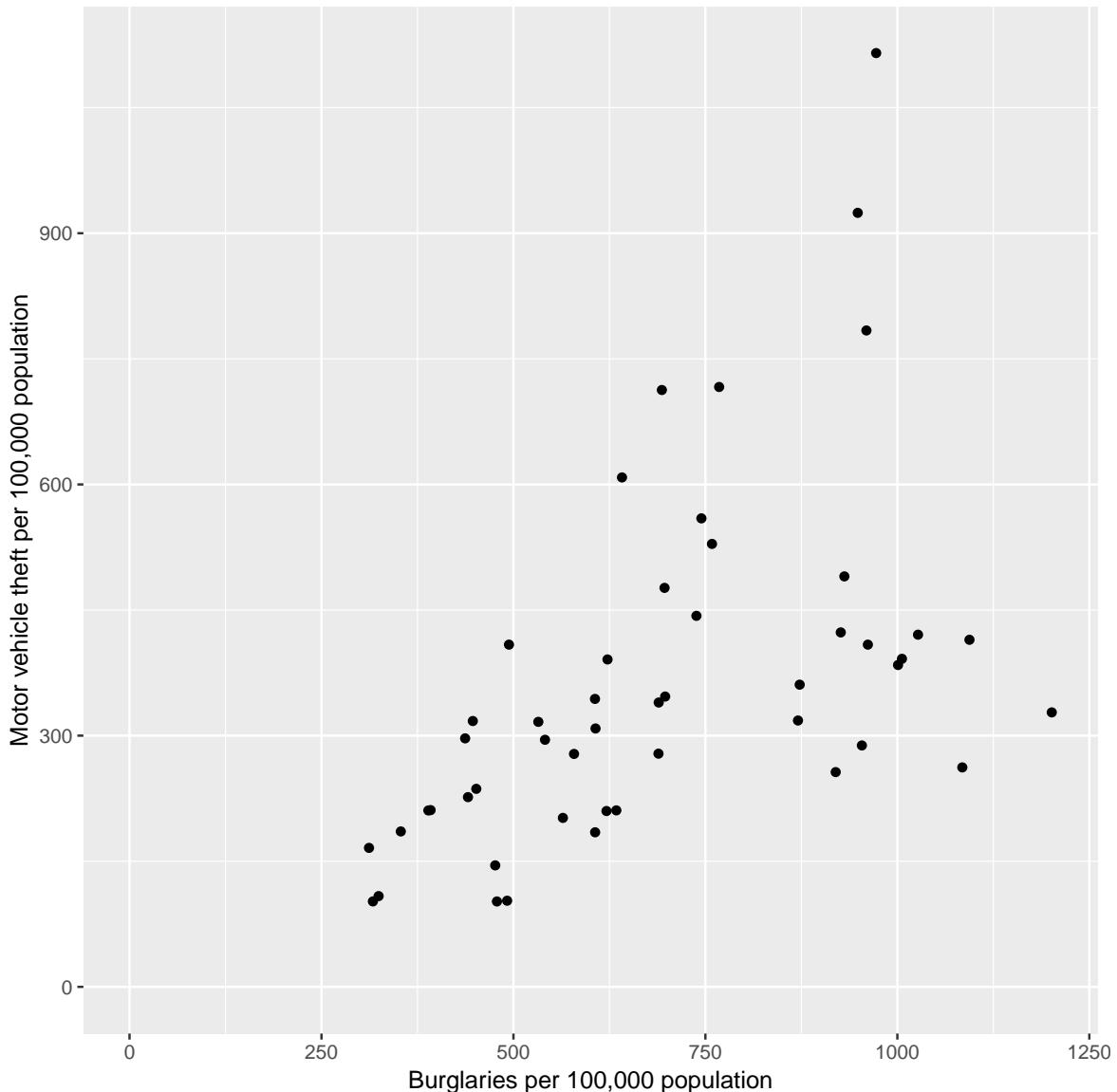


### 5.2.2 Customizing axes

`ggplot` also provides default axis extents (i.e., limits) and other axis features. These, and other axis features such as tick marks, labels, and transformations, can be changed using the scale functions. Here the range of the x and y axis is altered to start at zero

and go to the maximum of the x and y variables.<sup>3</sup> Here too, axis labels are specified within the scale function, which is an alterative to using the `labs()` function.

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft)) +
  geom_point() +
  scale_x_continuous(name="Burglaries per 100,000 population",
                     limits=c(0,max(crime$burglary))) +
  scale_y_continuous(name="Motor vehicle theft per 100,000 population",
                     limits = c(0, max(crime$motor_vehicle_theft)))
```




---

<sup>3</sup>ggplot2 makes the axes extend slightly beyond the given range, since typically this is what the user wants.

### 5.2.3 Text, point size, and color

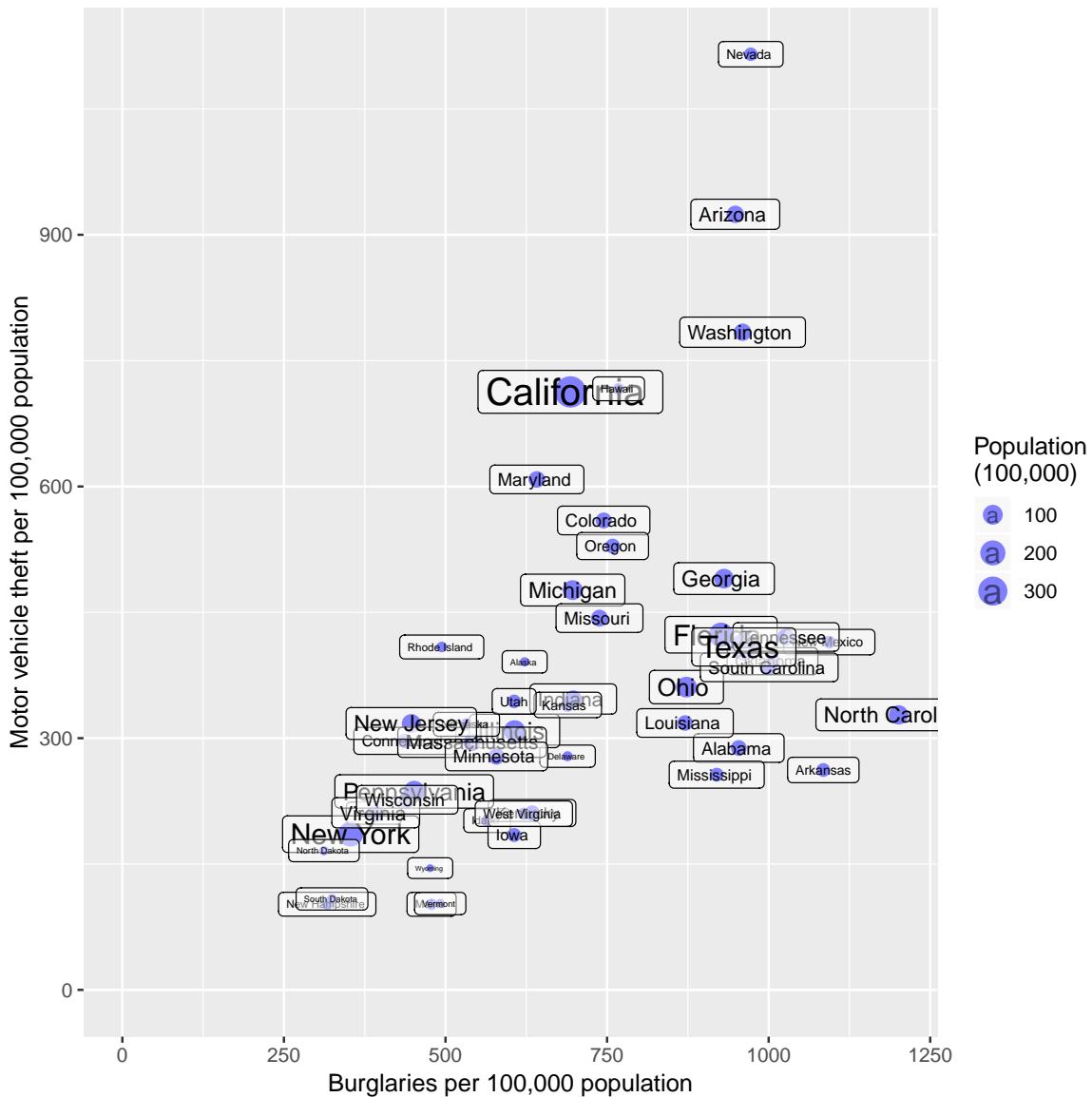
Next we make point size proportional to population, change the color, and add a state label. Note, in the `ggplot()` call I scaled population by 100,000 to help with the interpretability of the legend. Accordingly, I also changed the “population” label on the legend to “Population\n(100,000)” using the `labs()` function<sup>4</sup>. We use the `geom_label()` function to add the label, which provide an outline around the label text and allows you to control the box characteristics, e.g., I make the boxes slightly transparent using the `alpha` argument.<sup>5</sup>.

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft,
    size=population/100000)) +
  geom_point(color = "blue") +
  geom_label(aes(label = state), alpha = 0.5) +
  scale_x_continuous(name="Burglaries per 100,000 population",
    limits=c(0,max(crime$burglary))) +
  scale_y_continuous(name="Motor vehicle theft per 100,000 population",
    limits = c(0, max(crime$motor_vehicle_theft))) +
  labs(size="Population\n(100,000)")
```

---

<sup>4</sup>The \n is the line break and puts “(100,000)” below “Population”.

<sup>5</sup>You can also add labels via `geom_text()` function or a `label` argument in the `ggplot()` call.



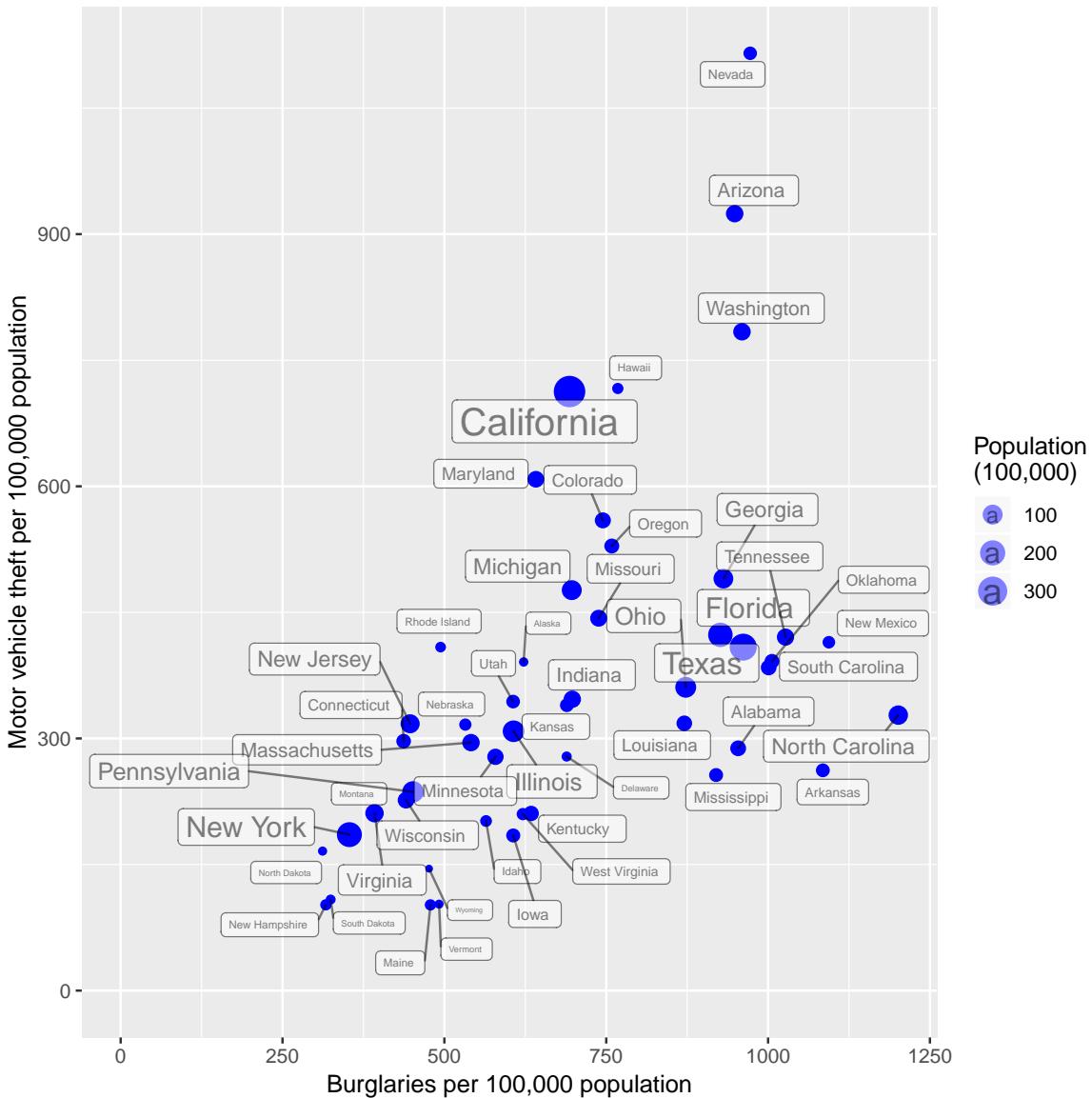
The labels are helpful but just too cluttered. There are some additional arguments that can go into `geom_label()` that allow for label offset; however, this won't help us much here. Instead, we can try the `ggrepel` package by Kamil Slowikowski. This useful package will automatically adjust labels so that they don't overlap. First we need to download and add the package using either RStudio's install package buttons or via `install.packages("ggrepel")`. Next to make all of `ggrepel`'s functions available we can call `library(ggrepel)` function or, if we know which function we want, we can load only that particular function using the `::` operators. I use `::` below to make clear which function is coming from `ggrepel` and which is coming from `ggplot2`.

```
ggplot(data = crime, aes(x = burglary, y = motor_vehicle_theft,
                         size=population/100000)) +
  geom_point(color = "blue") +
```

```

scale_x_continuous(name="Burglaries per 100,000 population",
                   limits=c(0,max(crime$burglary))) +
scale_y_continuous(name="Motor vehicle theft per 100,000 population",
                   limits = c(0, max(crime$motor_vehicle_theft))) +
labs(size="Population\n(100,000)") +
ggrepel::geom_label_repel(aes(label = state), alpha = 0.5)

```



This looks a bit better. We'll resist making additional improvements to the figure for now.

## 5.3 Other types of graphics

Scatter and line plots, which have just been presented, are common but certainly not the only graphical displays in common use. Histograms, boxplots, and bar graphs, as well as more “mathematical” displays such as the graph of a function, are commonly used to represent data. Examples of each are presented below.

### 5.3.1 Histograms

Simon Newcomb conducted several experiments to estimate the speed of light by measuring the time it took for light to travel from his laboratory to a mirror at the base of the Washington Monument, and then back to his lab. This is a distance of 7.44373 km, and by dividing this distance by the measured time, an estimate for the speed of light is obtained.

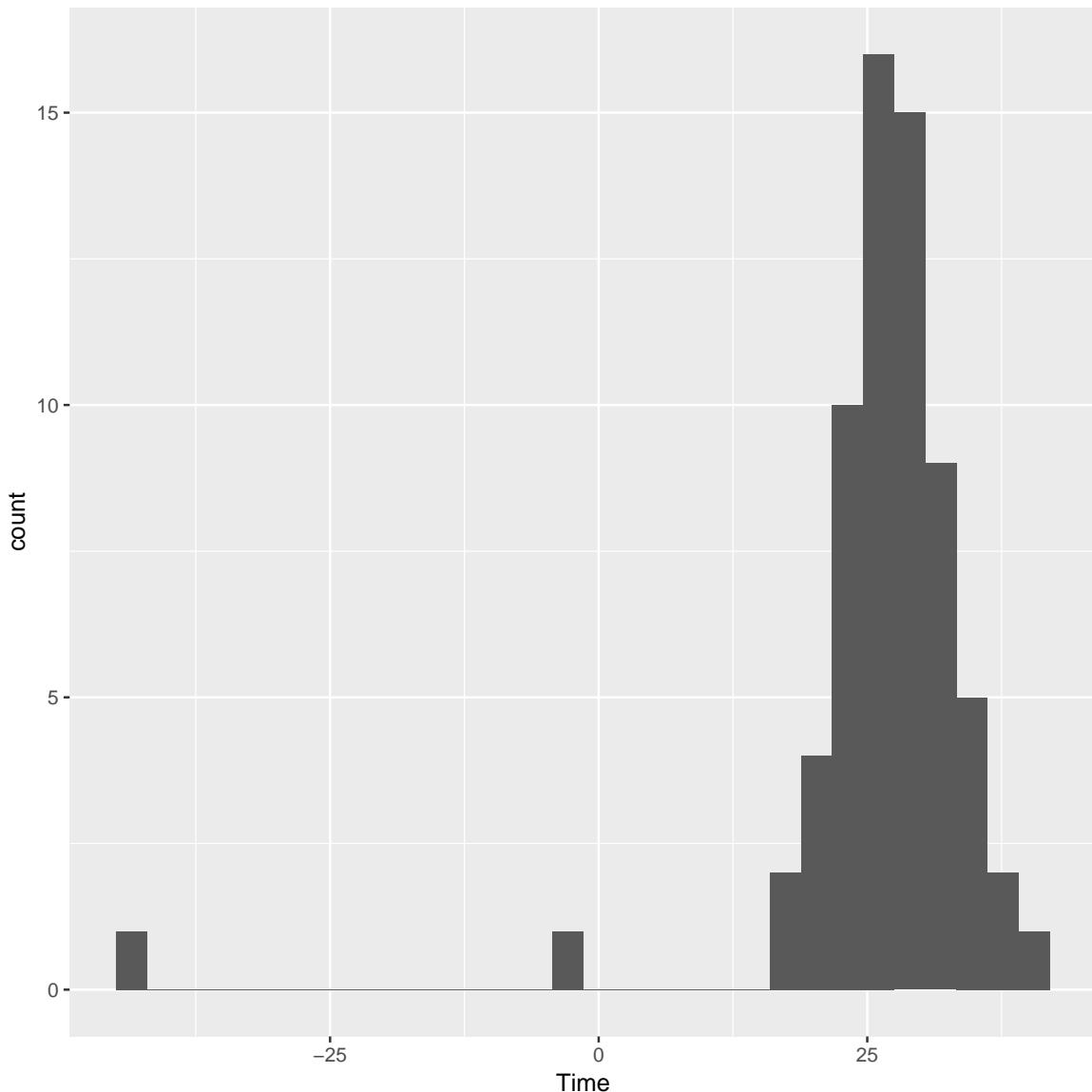
The times are of course quite small, and to avoid working with very small numbers, the data are recoded to be the deviation from 24800 nanoseconds. For example an observation coded as 28 represents a time of 24828 nanoseconds, while an observation coded as  $-44$  represents a time of 24756 nanoseconds.

```
> u.newcomb <- "http://blue.for.msu.edu/FOR875/data/Newcomb.csv"
> Newcomb <- read.csv(u.newcomb, header = TRUE)
> head(Newcomb)
```

|   | Time |
|---|------|
| 1 | 28   |
| 2 | 26   |
| 3 | 33   |
| 4 | 24   |
| 5 | 34   |
| 6 | -44  |

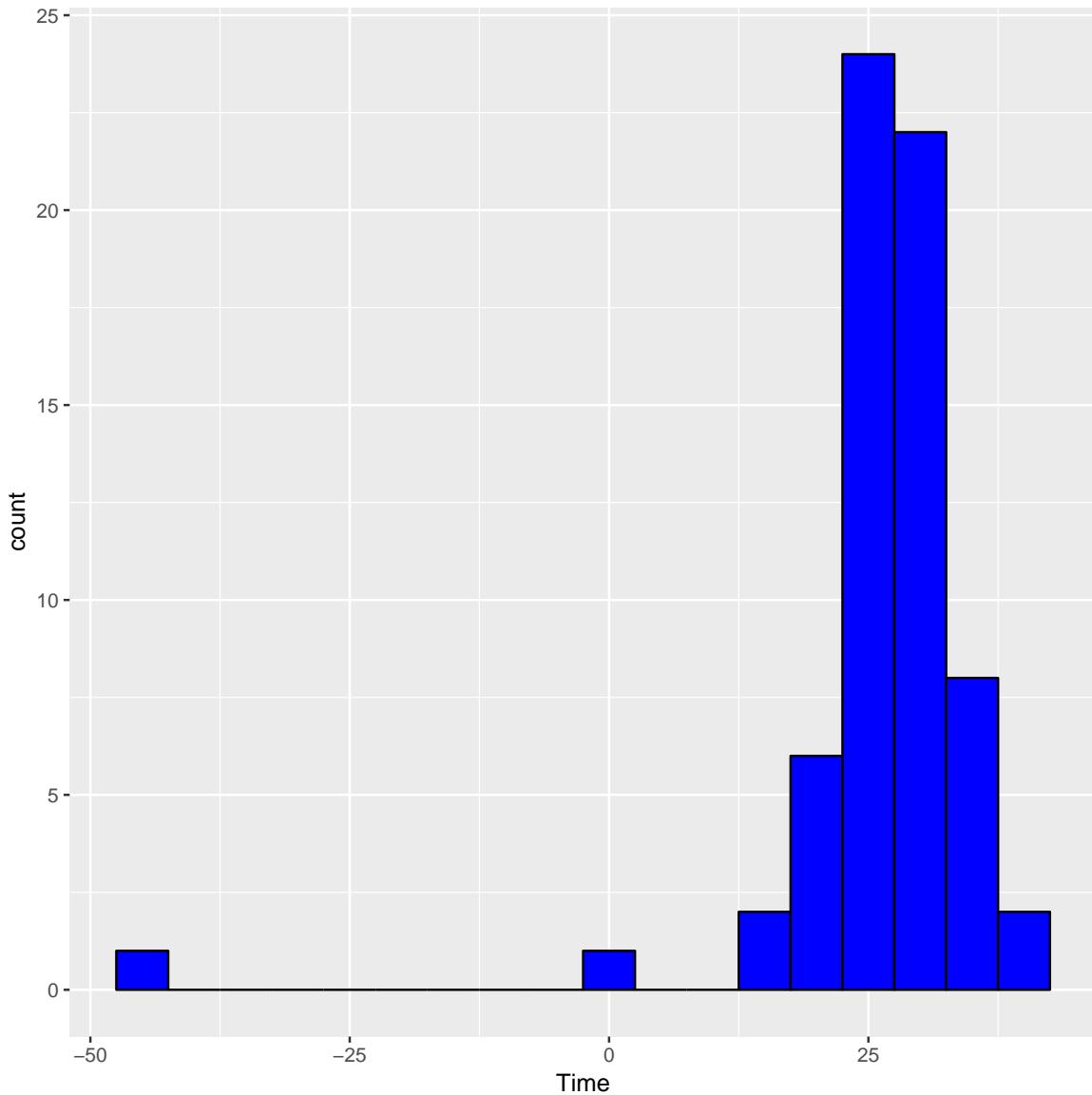
```
ggplot(Newcomb, aes(x = Time)) + geom_histogram()

'stat_bin()' using 'bins = 30'. Pick better value with
'binwidth'.
```



The software has an algorithm to calculate bin widths for the histogram. Sometimes the algorithm makes choices that aren't suitable (hence the R message above), and these can be changed by specifying a `binwidth`. In addition, the appearance of the bars also can be changed.

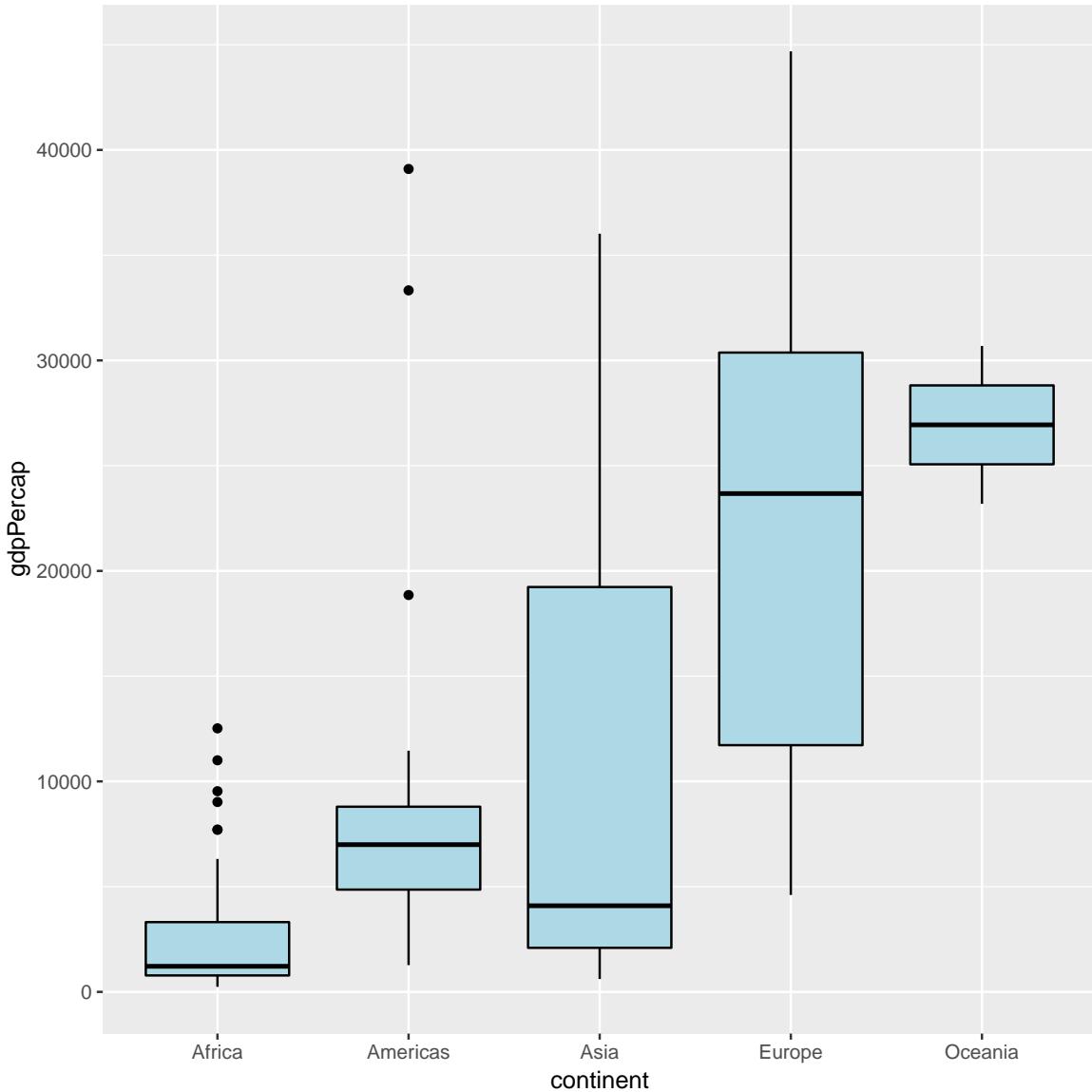
```
ggplot(Newcomb, aes(x = Time)) +  
  geom_histogram(binwidth = 5, color = "black", fill = "blue" )
```



### 5.3.2 Boxplots

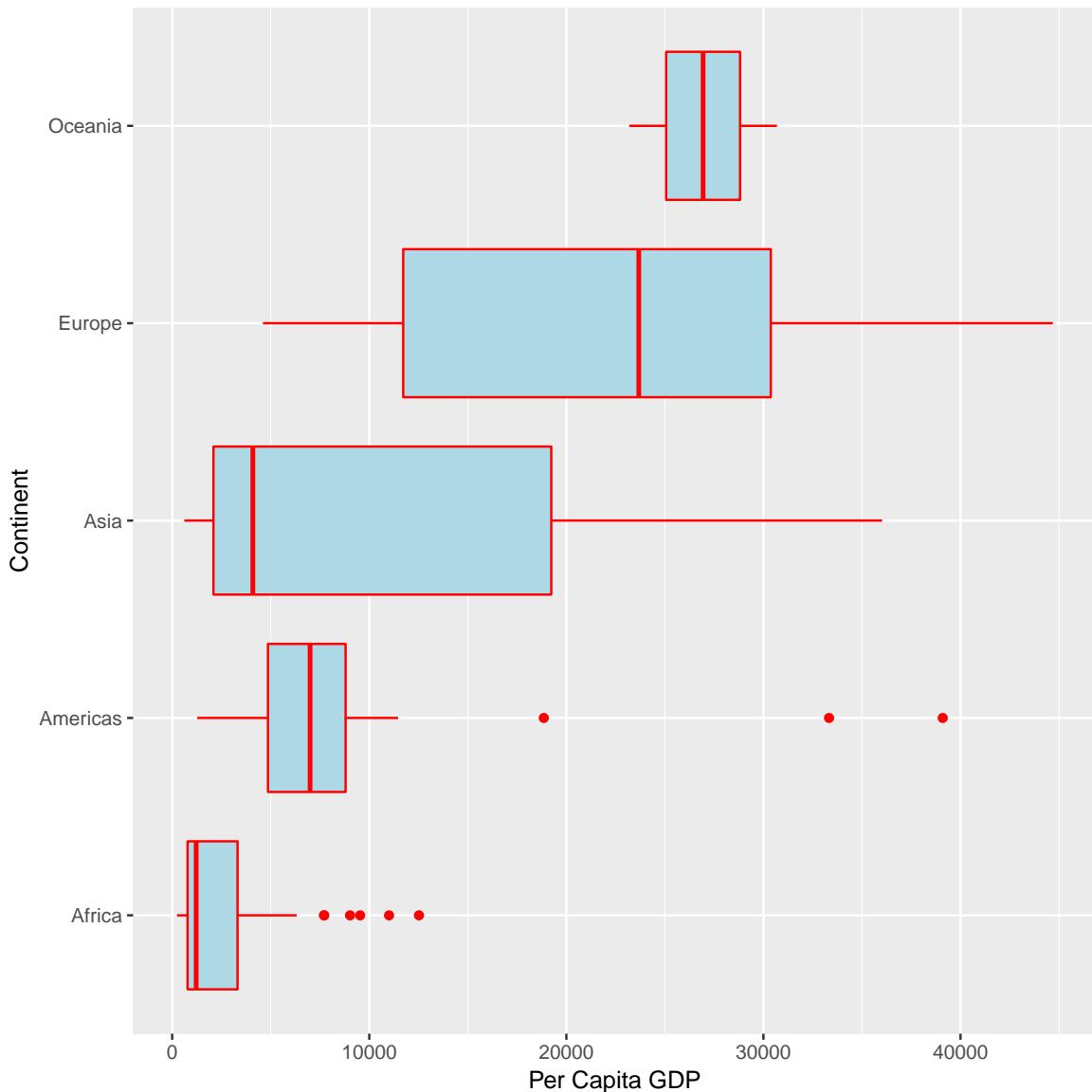
Next we consider some data from the gapminder data set to construct some box plots. These data are available in the `gapminder` package, which might need to be installed via `install.packages(gapminder)`.

```
library(gapminder)
ggplot(data = subset(gapminder, year == 2002),
       aes(x = continent, y = gdpPercap)) +
  geom_boxplot(color = "black", fill = "lightblue")
```



Here's the same set of boxplots, but with different colors, different axis labels, and the boxes plotted horizontally rather than vertically.

```
ggplot(data = subset(gapminder, year == 2002),
       aes(x = continent, y = gdpPercap)) +
  geom_boxplot(color = "red", fill = "lightblue") +
  scale_x_discrete(name = "Continent") +
  scale_y_continuous(name = "Per Capita GDP") + coord_flip()
```



### 5.3.3 Bar graphs

As part of a study, elementary school students were asked which was more important to them: good grades, popularity, or athletic ability. Here is a brief look at the data.

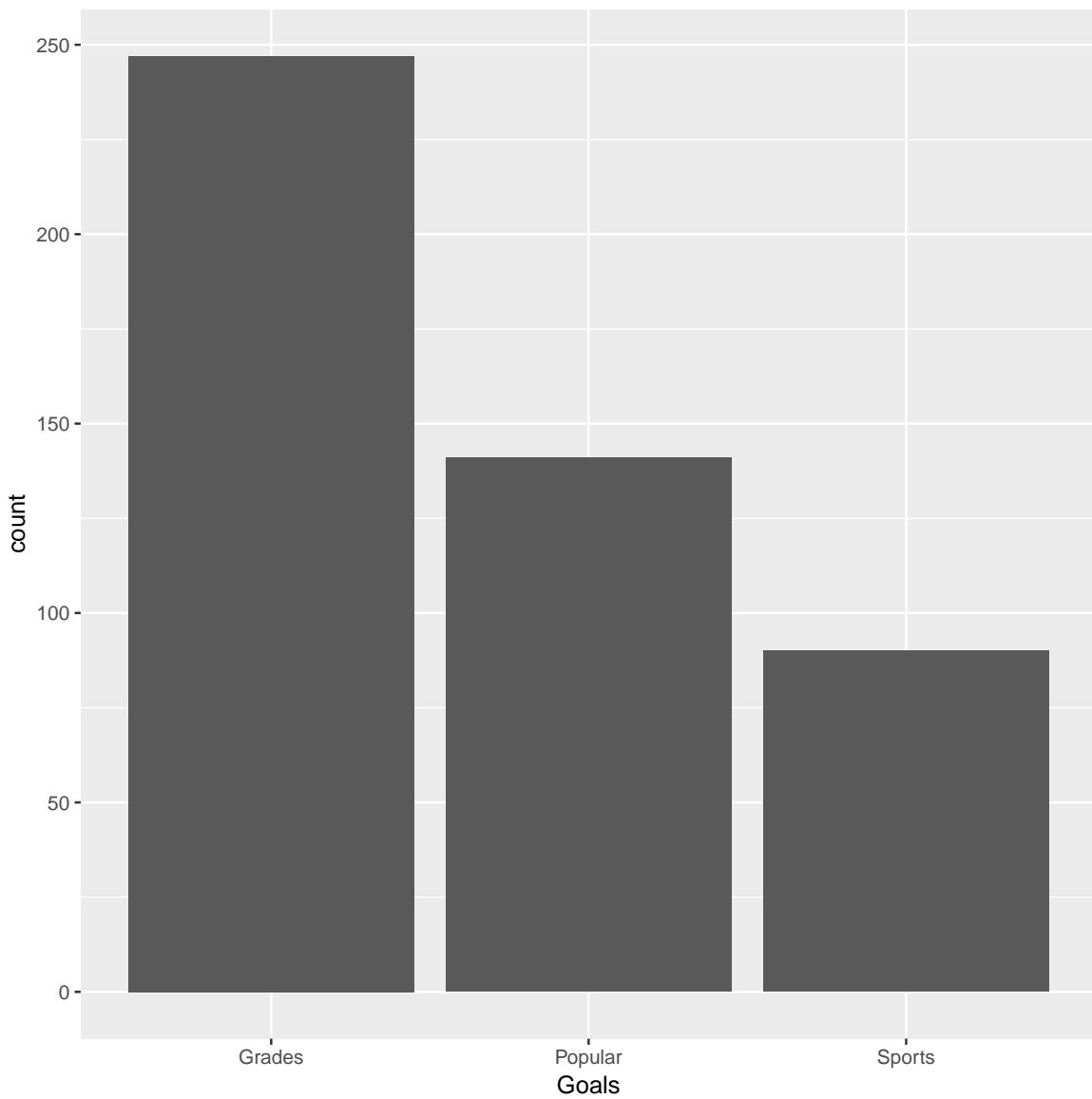
```
> u.goals <- "http://blue.for.msu.edu/FOR875/data/StudentGoals.csv"
> StudentGoals <- read.csv(u.goals, header = TRUE)
> head(StudentGoals)
```

|   | Gender | Grade | Age | Race  | Type  | School | Goals   | Grades | Sports |
|---|--------|-------|-----|-------|-------|--------|---------|--------|--------|
| 1 | boy    | 5     | 11  | White | Rural | Elm    | Sports  | 1      | 2      |
| 2 | boy    | 5     | 10  | White | Rural | Elm    | Popular | 2      | 1      |

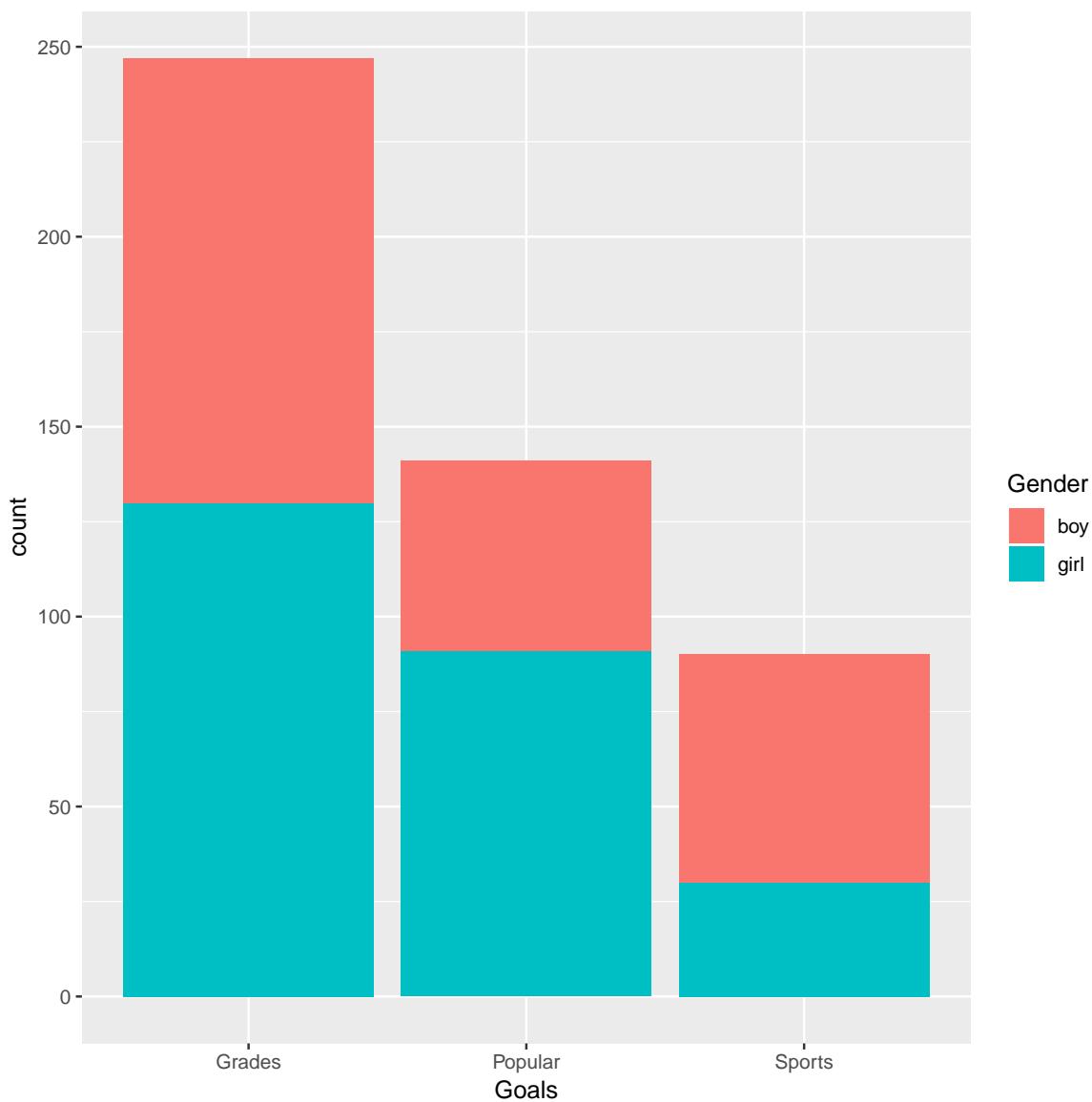
|   |      |       |       |       |       |     |         |   |   |
|---|------|-------|-------|-------|-------|-----|---------|---|---|
| 3 | girl | 5     | 11    | White | Rural | Elm | Popular | 4 | 3 |
| 4 | girl | 5     | 11    | White | Rural | Elm | Popular | 2 | 3 |
| 5 | girl | 5     | 10    | White | Rural | Elm | Popular | 4 | 2 |
| 6 | girl | 5     | 11    | White | Rural | Elm | Popular | 4 | 2 |
|   |      | Looks | Money |       |       |     |         |   |   |
| 1 |      | 4     | 3     |       |       |     |         |   |   |
| 2 |      | 4     | 3     |       |       |     |         |   |   |
| 3 |      | 1     | 2     |       |       |     |         |   |   |
| 4 |      | 4     | 1     |       |       |     |         |   |   |
| 5 |      | 1     | 3     |       |       |     |         |   |   |
| 6 |      | 1     | 3     |       |       |     |         |   |   |

First a simple bar graph of the most important goal chosen is drawn, followed by a stacked bar graph which also includes the student's gender, followed by a side by side bar graph which includes the student's gender.

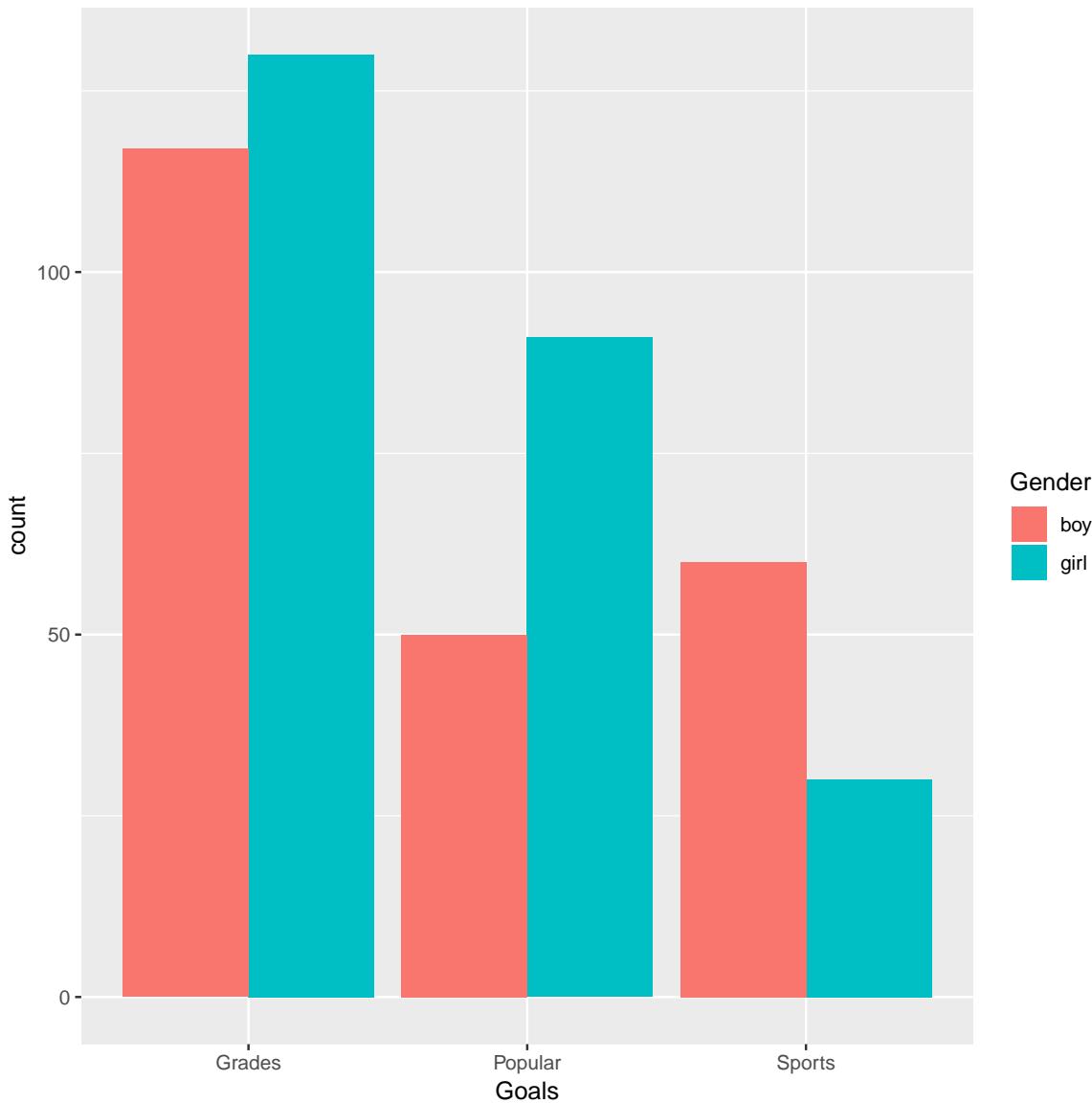
```
ggplot(StudentGoals, aes(x = Goals)) + geom_bar()
```



```
ggplot(StudentGoals, aes(x = Goals, fill = Gender)) + geom_bar()
```



```
ggplot(StudentGoals, aes(x = Goals, fill = Gender)) +  
  geom_bar(position = "dodge")
```

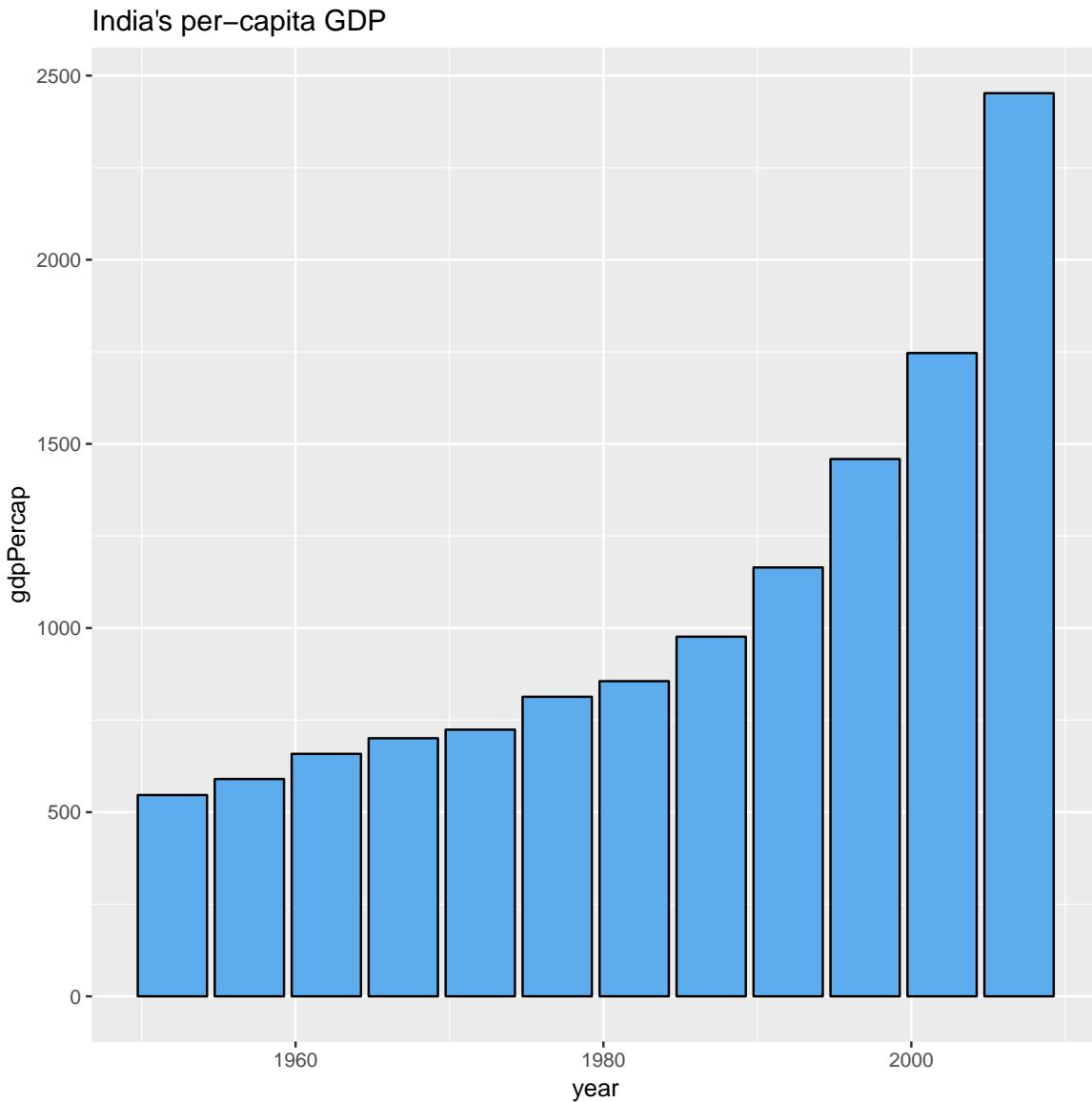


In this example R counted the number of students who had each goal and used these counts as the height of the bars. Sometimes the data contain the bar heights as a variable. For example, we create a bar graph of India's per capita GDP with separate bars for each year in the data<sup>6</sup>.

```
ggplot(subset(gapminder, country == "India"), aes(x = year, y = gdpPercap)) +
  geom_bar(stat = "identity", color = "black", fill = "steelblue2") +
  ggtitle("India's per-capita GDP")
```

---

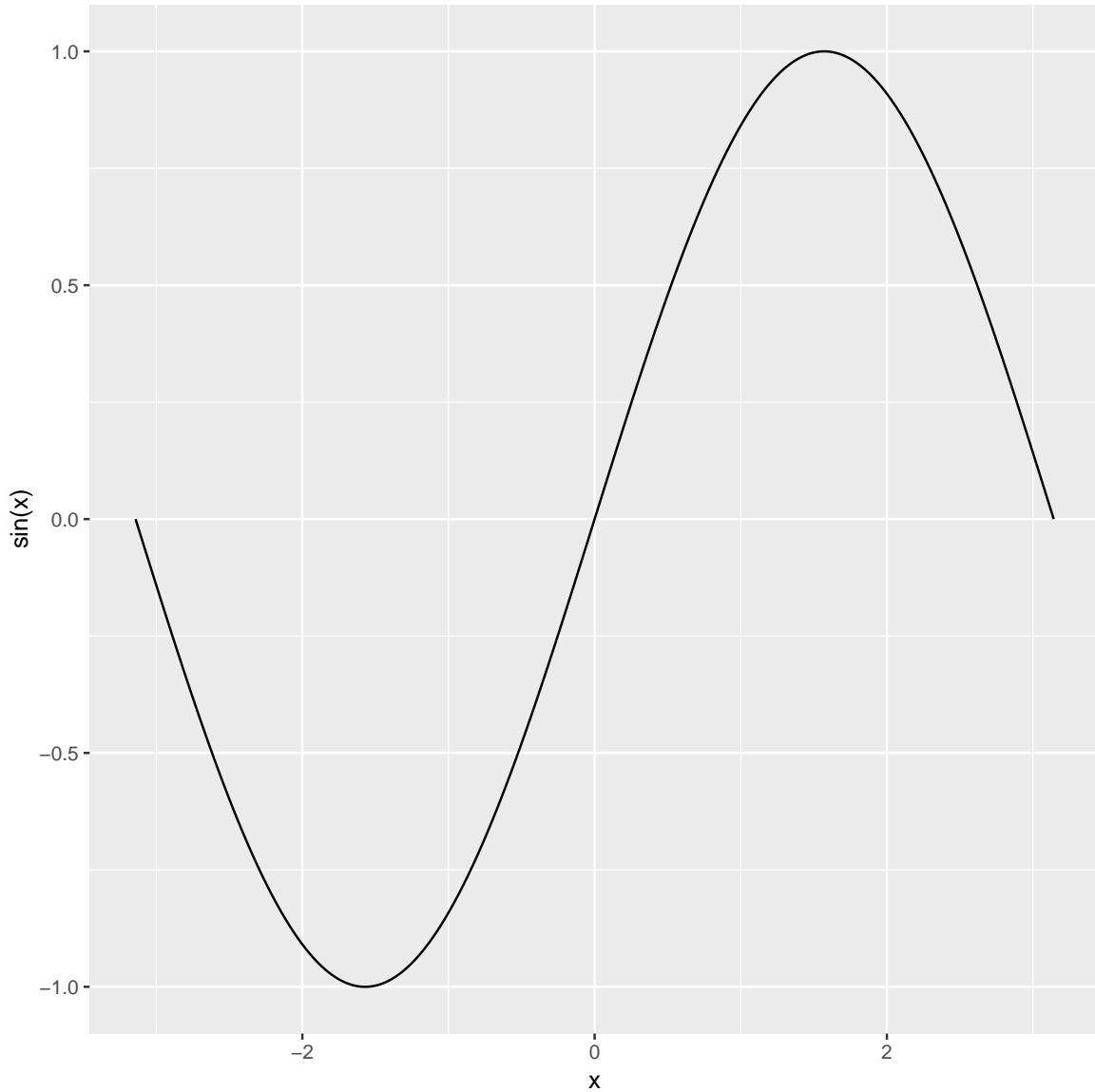
<sup>6</sup>R offers a large color palette, run `colors()` on the console to see a list of color names.



### 5.3.4 Graphs of functions

One way to create a plot of a mathematical function  $f$  is to create a data frame with  $x$  values in one column and  $f(x)$  values in another column, and then draw a line plot.

```
x <- seq(-pi, pi, len = 1000)
sin.data <- data.frame(x = x, y = sin(x))
ggplot(data = sin.data, aes(x = x, y = y)) + geom_line() +
  scale_y_continuous(name = "sin(x)")
```

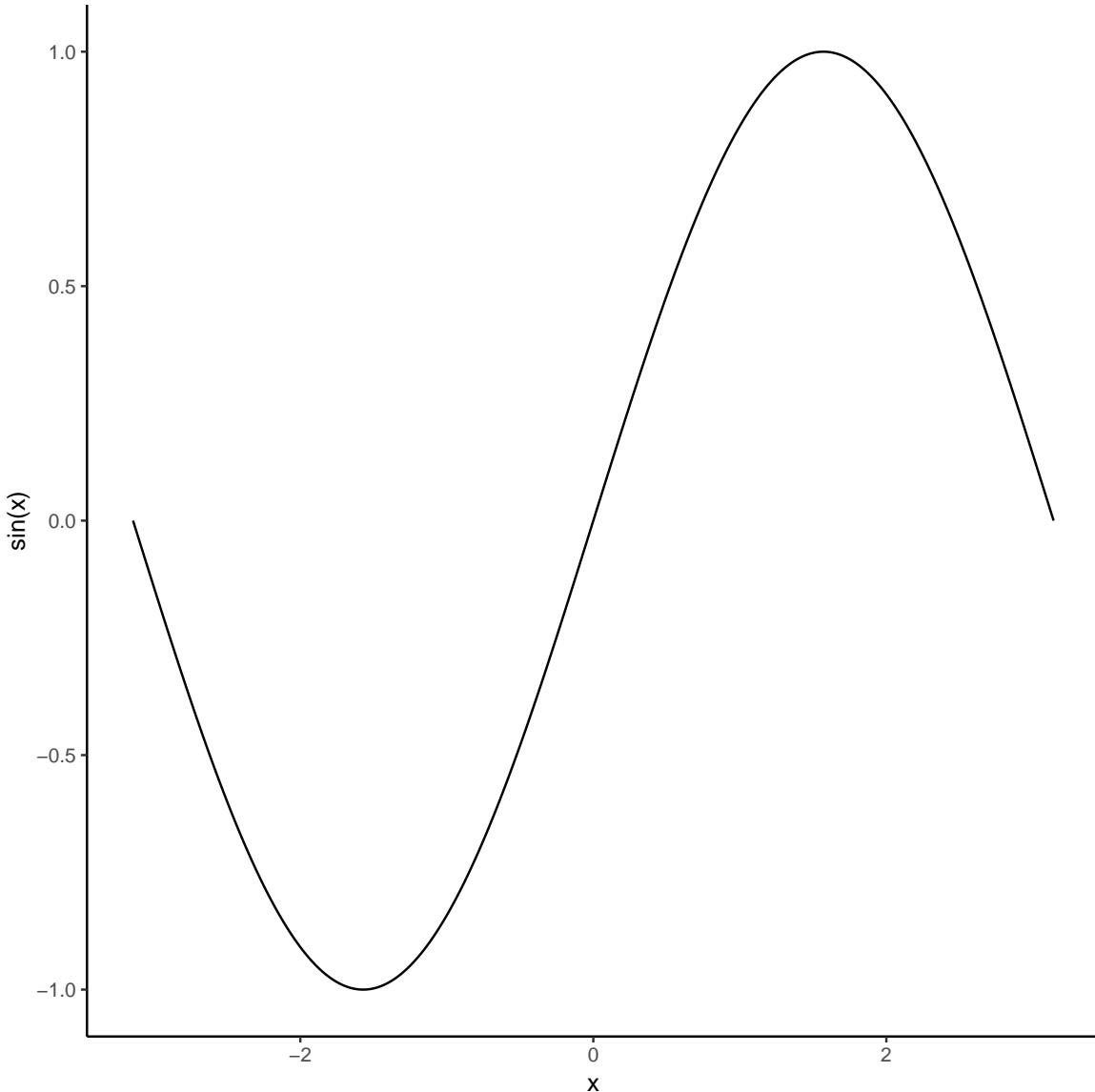


This method works well, but with a better understanding of functions in R we will be able to plot mathematical functions in a simpler and more natural way.

## 5.4 Themes

The theme defines non-data aspects of the plots characteristics such as background color, axes, and grid lines. Default themes include: `theme_bw()`, `theme_classic()`, `theme_dark()`, `theme_gray()`, `theme_light()`, `theme_linedraw()`, `theme_minimal()`, and `theme_void()`. Changing the theme is as easy as adding it to your initial `ggplot()` call. Here I replace the default implicit `theme_bw()` theme with the classic theme.

```
ggplot(data = sin.data, aes(x = x, y = y)) + geom_line() +
  scale_y_continuous(name = "sin(x)") +
  theme_classic()
```



The `ggthemes` add-on package <https://github.com/jrnold/ggthemes> by Jeffrey Arnold provides a large selection of themes beyond the eight themes that come with `ggplot2`.

## 5.5 Saving graphics

We often want to export our graphics to use in an external document or share with colleagues. There are several ways to save graphics in a variety of file formats. The `ggsave()` function will allow you to save your most recent `ggplot()` to a variety of

vector (e.g., “eps”, “ps”, “pdf”, “svg”) or raster (e.g., “jpeg”, “tiff”, “png”, “bmp”, “wmf”) formats<sup>7</sup>. The subsequent call to `ggsave()` saves the `sin.data` plot to a pdf file called “sin-plot.pdf”.

```
ggplot(filename = "sin-plot.pdf", device="pdf")
```

The `ggplot()` function takes additional arguments to control scale, measurement units, and raster plot resolution, i.e., dots per inch (dpi).

## 5.6 More resources

In summary, `ggplot2` provides a fairly intuitive<sup>8</sup> framework for developing an enormous variety of graphics. In addition to the resources mentioned at the beginning of this chapter, there are numerous online `ggplot2` resources and galleries to get ideas for creating beautiful graphics to convey the stories in your data. See, for example,

- <http://docs.ggplot2.org>
- <http://www.r-graph-gallery.com/portfolio/ggplot2-package>
- <http://www.ggplot2-exts.org/gallery>
- <http://www.cookbook-r.com/Graphs>
- and of course [www.google.com](http://www.google.com)

While the built-in `ggplot2` package documentation, accessible via the help tab in RStudio, is helpful, the official online documentation at <http://docs.ggplot2.org> is particularly useful because it provide example plots and navigation between related topics. The large number number of functions and syntax in `ggplot2` can be daunting. RStudio provides some handy cheatsheets to help you along [www.rstudio.com/resources/cheatsheets](http://www.rstudio.com/resources/cheatsheets) or direct link [www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf](http://www.rstudio.com/wp-content/uploads/2016/11/ggplot2-cheatsheet-2.1.pdf).

`ggplot2` also has an active mailing list at <http://groups.google.com/group/ggplot2>. The list is an excellent resource for users at all stages of experience. Another useful resource is stackoverflow, <http://stackoverflow.com>. There is an active `ggplot2` community on stackoverflow, and many common questions have already been asked and answered. When posting questions on any programming mailing list, it is best to provide a minimal reproducible example of your issue. The `reprex` <https://github.com/jennybc/reprex> package by Jenny Bryan provides a convenient way to do this, and also includes advice on creating a good example. The more information you provide about your issue, the more likely the community is to help you.

---

<sup>7</sup>Vector files comprise lines and curves known as paths, whereas raster files are comprised of pixels. Vector images are often preferred for publication quality graphics because they can be edited, scale well, and provide crisper detail.

<sup>8</sup>Like everything else in this book, it takes practice to get used to the syntax.

**Exercise 6**

Learning objectives: practice using `ggplot2` functions; summarize variables using graphics; introduce `ggplot2` facets.

# Chapter 6

## Working with Data

Bringing data into R, exporting data from R in a form that is readable by other software, cleaning and reshaping data, and other data manipulation tasks are an important and often overlooked component of data science. The book [Spector \[2008\]](#), while a few years old, is still an excellent reference for data-related issues. And the *R Data Import-/Export* manual, available online at <https://cran.r-project.org/doc/manuals/r-release/R-data.html>, is an up-to-date (and free) reference on importing a wide variety of datasets into R and on exporting data in various forms.

### 6.1 Reading data into R

Data come in a dizzying variety of forms. It might be in a proprietary format such as an `.xlsx` Excel file, a `.sav` SPSS file, or a `.mtw` Minitab file. It might be structured using a [relational model](#) comprising many tables that need to be connected via key-value pairs. It might be a data-interchange format such as [JSON](#) (JavaScript Object Notation), or markup language such [XML](#) (Extensible Markup Language) perhaps with specialized standards for describing ecological information, see, e.g., [EML](#) (Ecological Metadata Language). Both XML and EML are common data metadata formats (i.e., data that provides information about other data). Fortunately many datasets are (or can be) saved as plain text files, and most software can both read and write such files, so our initial focus will be on reading plain text files into R and saving data from R in plain text format. RStudio provides a handy [data import cheat sheet](#) for many of the read functions detailed in this section.

The `foreign` R package provides functions to directly read data saved in some of the proprietary formats into R, which is sometimes unavoidable, but if possible it is good to save data from another package as plain text and then read this plain text file into R. In Chapter 10 methods for reading web-based data sets into R will be discussed.

The function `read.table()` and its offshoots such as `read.csv()` are used to read in rectangular data from a text file. For example, the file `BrainAndBody.csv` contains

data<sup>1</sup> on the brain weight, body weight, and name of some terrestrial animals. Here are the first few lines of that file:

```
body,brain,name
1.35,8.1,Mountain beaver
465,423,Cow
36.33,119.5,Grey wolf
27.66,115,Goat
1.04,5.5,Guinea pig
```

As is evident, the first line of the file contains the names of the three variables, separated (delimited) by commas. Each subsequent line contains the body weight, brain weight, and name of a specific terrestrial animal.

This file is accessible at the url <http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv>. The `read.table()` function is used to read these data into an R data frame.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv"
> BrainBody <- read.table(file = u.bb, header = TRUE, sep = ",",
+                           stringsAsFactors = FALSE)
> head(BrainBody)

      body   brain         name
1     1.35    8.1 Mountain beaver
2    465.00  423.0           Cow
3    36.33 119.5       Grey wolf
4    27.66 115.0          Goat
5    1.04   5.5      Guinea pig
6 11700.00  50.0     Dipliodocus
```

The arguments used in this call to `read.table()` include:

1. `file = u.bb`, which tells R the location of the file. In this case the string <http://blue.for.msu.edu/FOR875/data/BrainAndBody.csv> giving the location is rather long, so it was first assigned to the object `u.bb`.
2. `header = TRUE`, which tells R the first line of the file gives the names of the variables.
3. `sep = ","`, which tells R that a comma separates the fields in the file.
4. `stringsAsFactors = FALSE` which tells R not to convert character vectors to factors.

---

<sup>1</sup>These data come from the MASS R library.

The function `read.csv()` is the same as `read.table()` except the default separator is a comma, whereas the default separator for `read.table()` is whitespace.

The file `BrainAndBody.tsv` contains the same data, except a tab is used in place of a comma to separate fields. The only change needed to read in the data in this file is in the `sep` argument (and of course the `file` argument, since the data are stored in a different file):

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.tsv"
> BrainBody2 <- read.table(file = u.bb, header = TRUE, sep = "\t",
+                           stringsAsFactors = FALSE)
> head(BrainBody2)

  body brain      name
1    1.35   8.1 Mountain beaver
2  465.00  423.0          Cow
3   36.33 119.5      Grey wolf
4   27.66 115.0         Goat
5    1.04   5.5    Guinea pig
6 11700.00   50.0     Dipliodocus
```

File extensions, e.g., `.csv` or `.tsv`, are naming conventions only and are there to remind us how the columns are separated, i.e., they have no influence on R's file read functions.

A third file, `BrainAndBody.txt`, contains the same data, but also contains a few lines of explanatory text above the names of the variables. It also uses whitespace rather than a comma or a tab as a separator. Here are the first several lines of the file.

```
This file contains data
on brain and body
weights of several terrestrial animals
```

```
"body" "brain" "name"
1.35 8.1 "Mountain beaver"
465 423 "Cow"
36.33 119.5 "Grey wolf"
27.66 115 "Goat"
1.04 5.5 "Guinea pig"
11700 50 "Dipliodocus"
2547 4603 "Asian elephant"
```

Notice that in this file the values of `name` are put inside of quotation marks. This is necessary since instead R would (reasonably) assume the first line contained the values of four variables, the values being 1.35, 8.1, Mountain, and beaver while in reality there are only three values desired, with Mountain beaver being the third.

To read in this file we need to tell R to skip the first five lines, and also that whitespace is the separator. The `skip` argument handles the first, and the `sep` argument the second. First let's see what happens if we don't use the `skip` argument.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.txt"
> BrainBody3 <- read.table(u.bb, header = TRUE, sep = " ",
+                           stringsAsFactors = FALSE)

Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec,
: line 1 did not have 5 elements
```

R assumed the first line of the file contained the variable names, since `header = TRUE` was specified, and counted four including `This`, `file`, `contains`, and `data`. So in the first line of actual data, R expected four columns containing data plus possibly a fifth column containing row names for the data set, and complained that “line 1 did not have 5 elements.” The error message is somewhat mysterious, since it starts with “Error in `scan`.” This happens because `read.table()` actually uses a more basic R function called `scan()` to do the work.

Here's how to read in the file correctly.

```
> u.bb <- "http://blue.for.msu.edu/FOR875/data/BrainAndBody.txt"
> BrainBody3 <- read.table(u.bb, header = TRUE, sep = " ",
+                           stringsAsFactors = FALSE, skip = 4)
> BrainBody3[1:10,]

      body   brain           name
1     1.35    8.1 Mountain beaver
2    465.00  423.0             Cow
3    36.33  119.5       Grey wolf
4    27.66  115.0            Goat
5     1.04    5.5 Guinea pig
6  11700.00   50.0 Dipliodocus
7  2547.00 4603.0 Asian elephant
8   187.10   419.0        Donkey
9   521.00   655.0          Horse
10    10.00   115.0 Potar monkey
```

### 6.1.1 Reading data with missing observations

Missing data is represented in many ways. Sometimes missing data is just that, i.e., the place where it should be in the file is blank. Other times specific numbers such as `-9999` or specific symbols are used. The `read.table()` function has an argument

`na.string` which allows the user to specify how missing data is indicated in the source file.

The site <http://www.wunderground.com/history> makes weather data available for locations around the world, and for dates going back to 1945. The file `WeatherKLAN2014.csv` contains weather data for Lansing, Michigan for the year 2014. Here are the first few lines of that file:

```
EST,Max TemperatureF,Min TemperatureF, Events
1/1/14,14,9,Snow
1/2/14,13,-3,Snow
1/3/14,13,-11,Snow
1/4/14,31,13,Snow
1/5/14,29,16,Fog-Snow
1/6/14,16,-12,Fog-Snow
1/7/14,2,-13,Snow
1/8/14,17,-1,Snow
1/9/14,21,2,Snow
1/10/14,39,21,Fog-Rain-Snow
1/11/14,41,32,Fog-Rain
1/12/14,39,31,
```

Look at the last line, and notice that instead of an Event such as `Snow` or `Fog-Snow` there is nothing after the comma. This observation is missing, but rather than using an explicit code such as `NA`, the site just leaves that entry blank. To read these data into R we will supply the argument `na.string = ""` which tells R the file indicates missing data by leaving the appropriate entry blank.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014.csv"
> WeatherKLAN2014 <- read.csv(u.weather, header=TRUE,
+                                     stringsAsFactors = FALSE, na.string = "")
```

|    | EST     | Max.TemperatureF | Min.TemperatureF | Events        |
|----|---------|------------------|------------------|---------------|
| 1  | 1/1/14  | 14               | 9                | Snow          |
| 2  | 1/2/14  | 13               | -3               | Snow          |
| 3  | 1/3/14  | 13               | -11              | Snow          |
| 4  | 1/4/14  | 31               | 13               | Snow          |
| 5  | 1/5/14  | 29               | 16               | Fog-Snow      |
| 6  | 1/6/14  | 16               | -12              | Fog-Snow      |
| 7  | 1/7/14  | 2                | -13              | Snow          |
| 8  | 1/8/14  | 17               | -1               | Snow          |
| 9  | 1/9/14  | 21               | 2                | Snow          |
| 10 | 1/10/14 | 39               | 21               | Fog-Rain-Snow |
| 11 | 1/11/14 | 41               | 32               | Fog-Rain      |

|    |         |    |    |           |
|----|---------|----|----|-----------|
| 12 | 1/12/14 | 39 | 31 | <NA>      |
| 13 | 1/13/14 | 44 | 34 | Rain      |
| 14 | 1/14/14 | 37 | 26 | Rain-Snow |
| 15 | 1/15/14 | 27 | 18 | Snow      |

## 6.2 Summarizing data frames

Some common data tasks include variable summaries such as means or standard deviations, transforming an existing variable, and creating new variables. As with many tasks, there are several ways to accomplish each of these.

### 6.2.1 Column (and row) summaries

The file `WeatherKLAN2014Full.csv` contains a more complete set of weather data variables than `WeatherKLAN2014.csv`, from the same source, <http://www.wunderground.com/history>.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014Full.csv"
> WeatherKLAN2014Full <- read.csv(u.weather, header=TRUE,
+                                     stringsAsFactors = FALSE,
+                                     na.string = "")
> names(WeatherKLAN2014Full)

[1] "EST"                      "Max.TemperatureF"
[3] "Mean.TemperatureF"        "Min.TemperatureF"
[5] "Max.Dew.PointF"           "MeanDew.PointF"
[7] "Min.DewpointF"            "Max.Humidity"
[9] "Mean.Humidity"             "Min.Humidity"
[11] "Max.Sea.Level.PressureIn" "Mean.Sea.Level.PressureIn"
[13] "Min.Sea.Level.PressureIn" "Max.VisibilityMiles"
[15] "Mean.VisibilityMiles"     "Min.VisibilityMiles"
[17] "Max.Wind.SpeedMPH"        "Mean.Wind.SpeedMPH"
[19] "Max.Gust.SpeedMPH"        "PrecipitationIn"
[21] "CloudCover"                "Events"
[23] "WindDirDegrees"
```

How can we compute the mean for each variable? One possibility is to do this a variable at a time:

```
> mean(WeatherKLAN2014Full$Mean.TemperatureF)
[1] 45.78082

> mean(WeatherKLAN2014Full$Min.TemperatureF)
[1] 36.25479

> mean(WeatherKLAN2014Full$Max.TemperatureF)
[1] 54.83836

> ## Et Cetera
```

This is pretty inefficient. Fortunately there is a `colMeans()` function which computes the mean of each (or specified) columns in a data frame. Some columns in the current data frame are not numeric, and obviously we don't want to ask R to compute means for these columns. We use `str()` to investigate.

```
> str(WeatherKLAN2014Full)

'data.frame': 365 obs. of 23 variables:
 $ EST                  : chr "2014-1-1" "2014-1-2" "2014-1-3" "2014-1-4" ...
 $ Max.TemperatureF    : int 14 13 13 31 29 16 2 17 21 39 ...
 $ Mean.TemperatureF   : int 12 5 1 22 23 2 -5 8 12 30 ...
 $ Min.TemperatureF   : int 9 -3 -11 13 16 -12 -13 -1 2 21 ...
 $ Max.Dew.PointF     : int 9 7 2 27 27 11 -6 7 18 37 ...
 $ MeanDew.PointF    : int 4 4 -5 18 21 -4 -13 1 8 28 ...
 $ Min.DewpointF      : int 0 -8 -14 3 11 -18 -18 -6 0 19 ...
 $ Max.Humidity        : int 88 76 83 92 92 80 78 88 88 100 ...
 $ Mean.Humidity       : int 76 70 68 73 86 73 72 78 75 92 ...
 $ Min.Humidity        : int 63 63 53 53 80 65 65 67 62 84 ...
 $ Max.Sea.Level.PressureIn: num 30.4 30.4 30.5 30.1 30 ...
 $ Mean.Sea.Level.PressureIn: num 30.3 30.2 30.4 30 29.9 ...
 $ Min.Sea.Level.PressureIn: num 30.2 30.1 30.1 29.9 29.7 ...
 $ Max.VisibilityMiles : int 10 9 10 10 4 10 10 10 10 9 ...
 $ Mean.VisibilityMiles: int 4 4 10 6 1 2 6 10 7 3 ...
 $ Min.VisibilityMiles: int 1 0 5 1 0 0 1 8 2 0 ...
 $ Max.Wind.SpeedMPH   : int 17 22 23 28 22 31 25 15 14 17 ...
 $ Mean.Wind.SpeedMPH  : int 9 13 10 15 11 18 15 7 6 10 ...
 $ Max.Gust.SpeedMPH  : int 22 30 32 36 30 40 31 18 17 22 ...
 $ PrecipitationIn     : chr "0.08" "0.01" "0.00" "0.12" ...
 $ CloudCover          : int 8 7 1 5 8 8 6 5 7 8 ...
```

```
$ Events : chr  "Snow" "Snow" "Snow" "Snow" ...
$ WindDirDegrees : int  43 24 205 203 9 262 220 236 147 160 ...
```

It isn't surprising that `EST` and `Events` are not numeric, but is surprising that `PrecipitationIn`, which measures precipitation in inches, also is not numeric, but is character. Let's investigate further.

```
> WeatherKLAN2014Full$PrecipitationIn[1:50]
```

```
[1] "0.08" "0.01" "0.00" "0.12" "0.78" "0.07" "T"      "T"
[9] "0.01" "0.39" "0.16" "0.00" "0.00" "0.01" "T"      "0.08"
[17] "T"      "T"      "T"      "0.01" "0.00" "0.05" "T"      "T"
[25] "0.07" "0.23" "0.04" "T"      "T"      "0.03" "T"      "0.37"
[33] "T"      "0.00" "T"      "0.27" "0.01" "T"      "0.04" "0.03"
[41] "T"      "0.00" "0.00" "T"      "T"      "0.00" "0.02" "0.15"
[49] "0.08" "0.01"
```

Now it's more clear. The original data file included `T` in the precipitation column to represent a “trace” of precipitation, which is precipitation greater than 0 but less than 0.01 inches. One possibility would be to set all these values to `"0"`, and then to convert the column to numeric. For now we will just leave the `PrecipitationIn` column out of the columns for which we request the mean.

```
> colMeans(WeatherKLAN2014Full[, c(2:19, 21, 23)])
```

|                           |                          |
|---------------------------|--------------------------|
| Max.TemperatureF          | Mean.TemperatureF        |
| 54.838356                 | 45.780822                |
| Min.TemperatureF          | Max.Dew.PointF           |
| 36.254795                 | 41.800000                |
| MeanDew.PointF            | Min.DewpointF            |
| 36.394521                 | 30.156164                |
| Max.Humidity              | Mean.Humidity            |
| 88.082192                 | 70.391781                |
| Min.Humidity              | Max.Sea.Level.PressureIn |
| 52.200000                 | 30.130247                |
| Mean.Sea.Level.PressureIn | Min.Sea.Level.PressureIn |
| 30.015370                 | 29.903890                |
| Max.VisibilityMiles       | Mean.VisibilityMiles     |
| 9.895890                  | 8.249315                 |
| Min.VisibilityMiles       | Max.Wind.SpeedMPH        |
| 4.824658                  | 19.101370                |
| Mean.Wind.SpeedMPH        | Max.Gust.SpeedMPH        |
| 8.679452                  | NA                       |

|            |                |
|------------|----------------|
| CloudCover | WindDirDegrees |
| 4.367123   | 205.000000     |

### 6.2.2 The apply() function

R also has functions `rowMeans()`, `colSums()`, and `rowSums()`. But what if we want to compute the median or standard deviation of columns of data, or some other summary statistic? For this the `apply()` function can be used. This function applies a user-chosen function to either the rows or columns (or both) of a data frame. The arguments are `X`, the data frame of interest, `MARGIN`, specifying either rows (`MARGIN = 1`) or columns (`MARGIN = 2`), and `FUN`, the function to be applied.

```
> apply(X = WeatherKLAN2014Full[, c(2:19, 21, 23)], MARGIN = 2,
+       FUN = sd)
```

|                           |                          |
|---------------------------|--------------------------|
| Max.TemperatureF          | Mean.TemperatureF        |
| 22.2129716                | 20.9729330               |
| Min.TemperatureF          | Max.Dew.PointF           |
| 20.2596676                | 19.5167159               |
| MeanDew.PointF            | Min.DewpointF            |
| 20.0311014                | 20.8511271               |
| Max.Humidity              | Mean.Humidity            |
| 8.1909784                 | 9.3660269                |
| Min.Humidity              | Max.Sea.Level.PressureIn |
| 13.9461681                | 0.2032259                |
| Mean.Sea.Level.PressureIn | Min.Sea.Level.PressureIn |
| 0.2159484                 | 0.2359542                |
| Max.VisibilityMiles       | Mean.VisibilityMiles     |
| 0.5790382                 | 2.1059259                |
| Min.VisibilityMiles       | Max.Wind.SpeedMPH        |
| 3.8168143                 | 6.4831246                |
| Mean.Wind.SpeedMPH        | Max.Gust.SpeedMPH        |
| 3.8862592                 | NA                       |
| CloudCover                | WindDirDegrees           |
| 2.7798359                 | 90.0673130               |

As with any R function the arguments don't need to be named as long as they are specified in the correct order, so

```
apply(WeatherKLAN2014Full[,c(2:19, 21, 23)], 2, sd)
```

has the same result<sup>2</sup>.

---

<sup>2</sup>See if you can figure out why `Max.Gust.SpeedMPH` is `NA` in the `apply()` output and make it return a numeric value. Hint, take a look at the `sd()` definition

### 6.2.3 Saving typing using with()

Consider calculating the mean of the maximum temperature values for those days where the cloud cover is less than 4 and when the maximum humidity is over 85. We can do this using subsetting.

```
> mean(WeatherKLAN2014Full$Max.TemperatureF[
+                               WeatherKLAN2014Full$CloudCover < 4 &
+                               WeatherKLAN2014Full$Max.Humidity > 85])
[1] 69.39241
```

While this works, it requires a lot of typing, since each time we refer to a variable in the data set we need to preface its name by `WeatherKLAN2014Full$`. The `with()` function tells R that we are working with a particular data frame, and we don't need to keep typing the name of the data frame.

```
> with(WeatherKLAN2014Full, mean(Max.TemperatureF[CloudCover <
+      4 & Max.Humidity > 85]))
[1] 69.39241
```

## 6.3 Transforming a data frame

Commonly variables are added to, removed from, changed in, or rearranged in, a data frame. The subsetting features of R make this reasonably easy. We will investigate this in the context of the `gapminder` data frame. If the `gapminder` library is not yet installed, use `install.packages("gapminder")` to install it locally.

```
> library(gapminder)
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679
 $ gdpPercap: num  779 821 853 836 740 ...
```

### 6.3.1 Adding variables

The data frame contains per capita GDP and population, and it might be interesting to create a variable that gives the total GDP by multiplying these two variables. (If we were interested in an accurate value for the total GDP we would probably be better off getting this information directly, since it is likely that the per capita GDP values in the data frame are rounded substantially.)

```
> gapminder$TotalGDP <- gapminder$gdpPercap * gapminder$pop
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 7 variables:
 $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num 28.8 30.3 32 34 36.1 ...
 $ pop      : int 8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679
 $ gdpPercap: num 779 821 853 836 740 ...
 $ TotalGDP : num 6.57e+09 7.59e+09 8.76e+09 9.65e+09 9.68e+09 ...
```

Analogous to the `with()` function, there is a function `within()` which can simplify the syntax. Whereas `with()` does not change the data frame, `within()` can. Note, below I first remove the altered gapminder dataframe using `rm()` then bring a clean copy back in by reloading the `gapminder` package.

```
> rm(gapminder)
> library(gapminder)
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 6 variables:
 $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num 28.8 30.3 32 34 36.1 ...
 $ pop      : int 8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679
 $ gdpPercap: num 779 821 853 836 740 ...

> gapminder <- within(gapminder, TotalGDP <- gdpPercap * pop)
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 7 variables:
 $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
```

```
$ lifeExp : num  28.8 30.3 32 34 36.1 ...
$ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679
$ gdpPercap: num  779 821 853 836 740 ...
$ TotalGDP : num  6.57e+09 7.59e+09 8.76e+09 9.65e+09 9.68e+09 ...
```

A nice feature of `within()` is its ability to add more than one variable at a time to a data frame. In this case the two or more formulas creating new variables must be enclosed in braces.

```
> gapminder <- within(gapminder, {TotalGDP <- gdpPercap * pop
+     lifeExpMonths <- lifeExp * 12})
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of  8 variables:
 $ country      : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent    : Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year         : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp       : num  28.8 30.3 32 34 36.1 ...
 $ pop          : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13
$ gdpPercap     : num  779 821 853 836 740 ...
$ TotalGDP      : num  6.57e+09 7.59e+09 8.76e+09 9.65e+09 9.68e+09 ...
$ lifeExpMonths: num  346 364 384 408 433 ...
```

### 6.3.2 Removing variables

After reflection we may realize the new variables we added to the `gapminder` data frame are not useful, and should be removed.

```
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of  8 variables:
 $ country      : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ continent    : Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 3 ...
 $ year         : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp       : num  28.8 30.3 32 34 36.1 ...
 $ pop          : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13
$ gdpPercap     : num  779 821 853 836 740 ...
$ TotalGDP      : num  6.57e+09 7.59e+09 8.76e+09 9.65e+09 9.68e+09 ...
$ lifeExpMonths: num  346 364 384 408 433 ...

> gapminder <- gapminder[1:6]
> str(gapminder)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 6 variables:
 $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 ...
 $ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ lifeExp  : num 28.8 30.3 32 34 36.1 ...
 $ pop      : int 8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679
 $ gdpPercap: num 779 821 853 836 740 ...
```

The same result could be obtained via `gapminder <- gapminder[, 1:6]`. The first method uses the fact that a data frame is also a list, and uses list subsetting methods. It is slightly preferable, since even if only one variable is retained, the object will still be a data frame, while the other method can return a vector in this case. Note this difference in the resulting `x` variable below (again this behavior can be frustrating at times, if it is not anticipated).

```
> a <- data.frame(x = 1:3, y = c("dog", "cat", "pig"), z = seq(from = 1,
+           to = 2, length = 3))
> a

  x   y   z
1 1 dog 1.0
2 2 cat 1.5
3 3 pig 2.0

> a <- a[1]
> a

  x
1 1
2 2
3 3

> a <- data.frame(x = 1:3, y = c("dog", "cat", "pig"), z = seq(from = 1,
+           to = 2, length = 3))
> a

  x   y   z
1 1 dog 1.0
2 2 cat 1.5
3 3 pig 2.0

> a <- a[, 1]
> a

[1] 1 2 3
```

One can also use a negative sign in front of the variable number(s). For example, `a[-(2:3)]` would drop the first two columns of `a`. Some care is needed when removing variables using the negative sign—what happens if you write `a[-2:3]` instead of `a[-(2:3)]` (why are the parentheses important here)?

An alternative approach is to set the variables you'd like to remove to `NULL`. For example, `a[c("y", "z")] <- NULL` and `a[,2:3] <- NULL` produce the same result as above.

### 6.3.3 Transforming variables

Consider the gapminder data again. Possibly we don't want to add a new variable that gives life expectancy in months, but rather want to modify the existing variable to measure life expectancy in months. Here are two ways to accomplish this.

```
> rm(gapminder)
> library(gapminder)
> gapminder$lifeExp[1:5]

[1] 28.801 30.332 31.997 34.020 36.088

> gapminder$lifeExp <- gapminder$lifeExp * 12
> gapminder$lifeExp[1:5]

[1] 345.612 363.984 383.964 408.240 433.056

> rm(gapminder)
> library(gapminder)
> gapminder$lifeExp[1:5]

[1] 28.801 30.332 31.997 34.020 36.088

> gapminder <- within(gapminder, lifeExp <- lifeExp * 12)
> gapminder$lifeExp[1:5]

[1] 345.612 363.984 383.964 408.240 433.056
```

### 6.3.4 Rearranging variables

Consider the full weather data set again.

```
> u.weather <- "http://blue.for.msu.edu/FOR875/data/WeatherKLAN2014Full.csv"
> WeatherKLAN2014Full <- read.csv(u.weather, header=TRUE,
+                                     stringsAsFactors = FALSE,
+                                     na.string = "")
> names(WeatherKLAN2014Full)

[1] "EST"                      "Max.TemperatureF"
[3] "Mean.TemperatureF"        "Min.TemperatureF"
[5] "Max.Dew.PointF"           "MeanDew.PointF"
[7] "Min.DewpointF"            "Max.Humidity"
[9] "Mean.Humidity"             "Min.Humidity"
[11] "Max.Sea.Level.PressureIn" "Mean.Sea.Level.PressureIn"
[13] "Min.Sea.Level.PressureIn" "Max.VisibilityMiles"
[15] "Mean.VisibilityMiles"     "Min.VisibilityMiles"
[17] "Max.Wind.SpeedMPH"        "Mean.Wind.SpeedMPH"
[19] "Max.Gust.SpeedMPH"        "PrecipitationIn"
[21] "CloudCover"                "Events"
[23] "WindDirDegrees"
```

If we want the wind speed variables to come right after the date, we can again use subsetting.

```
> WeatherKLAN2014Full <- WeatherKLAN2014Full[c(1, 17, 18, 19, 2:16,
+                                              20:23)]
> names(WeatherKLAN2014Full)

[1] "EST"                      "Max.Wind.SpeedMPH"
[3] "Mean.Wind.SpeedMPH"        "Max.Gust.SpeedMPH"
[5] "Max.TemperatureF"          "Mean.TemperatureF"
[7] "Min.TemperatureF"          "Max.Dew.PointF"
[9] "MeanDew.PointF"            "Min.DewpointF"
[11] "Max.Humidity"              "Mean.Humidity"
[13] "Min.Humidity"               "Max.Sea.Level.PressureIn"
[15] "Mean.Sea.Level.PressureIn" "Min.Sea.Level.PressureIn"
[17] "Max.VisibilityMiles"       "Mean.VisibilityMiles"
[19] "Min.VisibilityMiles"       "PrecipitationIn"
[21] "CloudCover"                 "Events"
[23] "WindDirDegrees"
```

## 6.4 Reshaping data

A data set can be represented in several different formats. Consider a (fictitious) data set on incomes of three people during three different years. Here is one representation of the data:

```
> yearlyIncomeWide

      name income1990 income2000 income2010
1 John Smith      29784      39210      41213
2 Jane Doe       56789      89321     109321
3 Albert Jones    2341      34567      56781
```

Here is another representation of the same data:

```
> yearlyIncomeLong

      name      year income
1 John Smith income1990  29784
2 Jane Doe  income1990  56789
3 Albert Jones income1990  2341
4 John Smith income2000  39210
5 Jane Doe   income2000  89321
6 Albert Jones income2000  34567
7 John Smith income2010  41213
8 Jane Doe   income2010 109321
9 Albert Jones income2010  56781
```

For hopefully obvious reasons, the first representation is called a *wide* representation of the data, and the second is called a *long* representation. Each has its merits. The first representation is probably easier for people to read, while the second is often the form needed for analysis by statistical software such as R. There are of course other representations. For example the rows and columns could be interchanged to create a different wide representation, or the long representation, which currently groups data by year, could group by name instead.

Whatever the relative merits of wide and long representations of data, transforming data from wide to long or long to wide is often required. As with many tasks, there are several ways to accomplish this in R. We will focus on a relatively new library called `tidyR` written by Hadley Wickham that performs the transformations and more.

### 6.4.1 TidyR

The R library `tidyR` has functions for converting data between formats. To illustrate its use, we examine a simple data set that explores the relationship between religion

and income in the United States. The data come from a Pew survey, and are used in the `tidyR` documentation to illustrate transforming data from wide to long format.

```
> u.rel <- "http://blue.for.msu.edu/FOR875/data/religion2.csv"
> religion <- read.csv(u.rel, header = TRUE, stringsAsFactors = FALSE)
> head(religion)

      religion under10k btw10and20k btw20and30k
1      Agnostic      27        34        60
2      Atheist       12        27        37
3      Buddhist      27        21        30
4      Catholic      418       617       732
5 DoNotKnowOrRefused    15        14        15
6 EvangelicalProt     575       869      1064
      btw30and40k btw40and50k btw50and75k btw75and100k
1          81        76       137       122
2          52        35        70        73
3          34        33        58        62
4         670       638      1116      949
5          11        10        35        21
6         982       881      1486      949
      btw100and150k over150k DoNotKnowOrRefused
1          109       84        96
2          59        74        76
3          39        53        54
4         792       633      1489
5          17        18        116
6         723       414      1529
```

As given, the columns include religion and income level, and there are counts for each of the combinations of religion and income level. For example, there are 27 people who are Agnostic and whose income is less than 10 thousand dollars, and there are 617 people who are Catholic and whose income is between 10 and 20 thousand dollars.

The `gather()` function can transform data from wide to long format.

```
> library(tidyR)
> religionLong <- gather(data = religion, key = IncomeLevel,
+                           value = Frequency, 2:11)
> head(religionLong)

      religion IncomeLevel Frequency
1      Agnostic   under10k        27
2      Atheist    under10k        12
3      Buddhist   under10k        27
```

```

4      Catholic    under10k     418
5 DoNotKnowOrRefused    under10k      15
6   EvangelicalProt    under10k    575

> tail(religionLong)

            religion      IncomeLevel Frequency
175          Muslim DoNotKnowOrRefused      22
176        Orthodox DoNotKnowOrRefused      73
177  OtherChristian DoNotKnowOrRefused      18
178    OtherFaiths DoNotKnowOrRefused      71
179 OtherWorldReligions DoNotKnowOrRefused      8
180      Unaffiliated DoNotKnowOrRefused    597

```

To use `gather()` we specified the data frame (`data = religion`), the name we want to give to the column created from the income levels (`key = IncomeLevel`), the name we want to give to the column containing the frequency values (`value = Frequency`) and the columns to gather (`2:11`).

Columns to be gathered can be specified by name also, and we can also specify which columns should be omitted using a negative sign in front of the name(s). So the following creates an equivalent data frame:

```

> religionLong <- gather(data = religion, key = IncomeLevel,
+                           value = Frequency, -religion)
> head(religionLong)

            religion IncomeLevel Frequency
1      Agnostic    under10k      27
2       Atheist    under10k      12
3     Buddhist    under10k      27
4      Catholic    under10k     418
5 DoNotKnowOrRefused    under10k      15
6   EvangelicalProt    under10k    575

```

```

> religionWide <- spread(data = religionLong, key = IncomeLevel,
+                           value = Frequency)
> head(religionWide)

            religion btw100and150k btw10and20k btw20and30k
1      Agnostic         109         34         60
2       Atheist          59         27         37
3     Buddhist          39         21         30

```

|   |  |      |      |      |
|---|--|------|------|------|
| 4 | Catholic   | 792  | 617  | 732  |
| 5 | DoNotKnowOrRefused                               | 17   | 14   | 15   |
| 6 | EvangelicalProt                                  | 723  | 869  | 1064 |
|   | btw30and40k btw40and50k btw50and75k btw75and100k |      |      |      |
| 1 | 81   | 76   | 137  | 122  |
| 2 | 52   | 35   | 70   | 73   |
| 3 | 34   | 33   | 58   | 62   |
| 4 | 670  | 638  | 1116 | 949  |
| 5 | 11   | 10   | 35   | 21   |
| 6 | 982  | 881  | 1486 | 949  |
|   | DoNotKnowOrRefused over150k under10k             |      |      |      |
| 1 |  | 96   | 84   | 27   |
| 2 |  | 76   | 74   | 12   |
| 3 |  | 54   | 53   | 27   |
| 4 |  | 1489 | 633  | 418  |
| 5 |  | 116  | 18   | 15   |
| 6 |  | 1529 | 414  | 575  |

Here we specify the data frame (`religionLong`), the column (`IncomeLevel`) to be spread, and the column of values (`Frequency`) to be spread among the newly created columns. As can be seen, this particular call to `spread()` yields the original data frame.

`tidyverse` provides two other useful functions to separate and unite variables based on some delimiter. Consider again the `yearlyIncomeWide` table. Say we want to split the `name` variable into first and last name. This can be done using the `separate()` function.

```
> firstLast <- separate(data = yearlyIncomeLong, col = name,
+                         into = c("first", "last"), sep="\s")
> print(firstLast)

  first   last      year income
1 John Smith income1990  29784
2 Jane Doe income1990  56789
3 Albert Jones income1990   2341
4 John Smith income2000  39210
5 Jane Doe income2000  89321
6 Albert Jones income2000  34567
7 John Smith income2010  41213
8 Jane Doe income2010 109321
9 Albert Jones income2010  56781
```

Now say, you're not happy with that and you want to combine the name column again, but this time separate the first and last name with a underscore. This is done

using the `unite()` function.

```
> unite(firstLast, col = name, first, last, sep = "_")
```

|   | name         | year       | income |
|---|--------------|------------|--------|
| 1 | John_Smith   | income1990 | 29784  |
| 2 | Jane_Doe     | income1990 | 56789  |
| 3 | Albert_Jones | income1990 | 2341   |
| 4 | John_Smith   | income2000 | 39210  |
| 5 | Jane_Doe     | income2000 | 89321  |
| 6 | Albert_Jones | income2000 | 34567  |
| 7 | John_Smith   | income2010 | 41213  |
| 8 | Jane_Doe     | income2010 | 109321 |
| 9 | Albert_Jones | income2010 | 56781  |

## 6.5 Manipulating data with dplyr

Much of the effort (a figure of 80% is sometimes suggested) in data analysis is spent cleaning the data and getting it ready for analysis. Having effective tools for this task can save substantial time and effort. The R package `dplyr` written by Hadley Wickham is designed, in Hadley’s words, to be “a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.” Casting data analysis tasks in terms of “grammar” should be familiar from our work with the `ggplot2` package, which was also authored by Hadley. Functions provided by `dplyr` do in fact capture key data analysis actions (i.e., verbs) including

`mutate()` adds new variables that are functions of existing variables,  
`select()` picks variables based on their names,  
`filter()` picks cases based on their values,  
`summarize()` reduces multiple values down to a single summary,  
`arrange()` changes the ordering of the rows.

These all combine naturally with a `group_by()` that allows you to perform any operation grouped by values of one or more variables. All the tasks done using `dplyr` can be accomplished using tools already covered in this text; however, `dplyr`'s functions provide a potentially more efficient and convenient framework to accomplish these tasks. RStudio provides a convenient [data wrangling cheat sheet](#) that covers many aspects of the `tidyverse` and `dplyr` packages.

This somewhat long section on `dplyr` adapts the nice introduction by Jenny Bryan, available at [http://stat545-ubc.github.io/block010\\_dplyr-end-single-table.html](http://stat545-ubc.github.io/block010_dplyr-end-single-table.html).

### 6.5.1 Improved data frames

The `dplyr` package provides two functions that offer improvements on data frames. First, `data_frame()` is a trimmed down version of the `data.frame()` function that is somewhat more user friendly, and won't be discussed here. Second, `tbl_df()` creates a tibble<sup>3</sup>. A tibble has two advantages over a data frame. First, when printing, it only prints the first ten rows and the columns that fit on the page, as well as some additional information about the table's dimension, data type of variables, and non-printed columns. Second, recall that subsetting a data frame can sometimes return a vector rather than a data frame (if only one row or column is the result of the subset), a tibble does not have this behavior. Here is an example using the `religionWide` data frame<sup>4</sup>.

```
> library(dplyr)

Attaching package: 'dplyr'
The following objects are masked from 'package:stats':
    filter, lag
The following objects are masked from 'package:base':
    intersect, setdiff, setequal, union

> head(religionWide)

      religion btw100and150k btw10and20k btw20and30k
1      Agnostic        109        34        60
2      Atheist          59        27        37
3      Buddhist         39        21        30
4      Catholic         792       617       732
5 DoNotKnowOrRefused      17        14        15
6 EvangelicalProt        723       869      1064
      btw30and40k btw40and50k btw50and75k btw75and100k
1          81        76        137       122
2          52        35        70        73
3          34        33        58        62
4         670       638       1116      949
5          11        10        35        21
6         982       881       1486      949
```

<sup>3</sup>Reminds me of [The Trouble with Tribbles](#).

<sup>4</sup>The text printed immediately after `library(dplyr)` means the `stats` and `base` packages, which are automatically loaded when you start R, have functions with the same name as functions in `dplyr`. So, for example, if you call the `filter()` or `lag()`, R will use `library(dplyr)`'s functions. Use the `::` operator to explicitly identify which packages' function you want to use, e.g., if you want `stats`'s `lag()` then call `stats::lag()`.

```

DoNotKnowOrRefused over150k under10k
1             96      84     27
2              76      74     12
3              54      53     27
4            1489     633    418
5              116     18     15
6            1529     414    575

> religionWide[, 1]

[1] "Agnostic"           "Atheist"
[3] "Buddhist"            "Catholic"
[5] "DoNotKnowOrRefused" "EvangelicalProt"
[7] "Hindu"                "HistoricallyBlackProt"
[9] "JehovahsWitness"     "Jewish"
[11] "MainlineProt"        "Mormon"
[13] "Muslim"               "Orthodox"
[15] "OtherChristian"      "OtherFaiths"
[17] "OtherWorldReligions" "Unaffiliated"

> religionWideTbl <- tbl_df(religionWide)
> head(religionWideTbl)

# A tibble: 6 x 11
  religion btw100and150k btw10and20k btw20and30k btw30and40k
  <chr>       <int>       <int>       <int>       <int>
1 Agnostic     109        34         60         81
2 Atheist       59        27         37         52
3 Buddhist      39        21         30         34
4 Catholic      792       617       732       670
5 DoNotKn~      17         14         15         11
6 Evangel~     723       869      1064      982
# ... with 6 more variables: btw40and50k <int>,
#   btw50and75k <int>, btw75and100k <int>,
#   DoNotKnowOrRefused <int>, over150k <int>,
#   under10k <int>

> religionWideTbl[, 1]

# A tibble: 18 x 1
  religion
  <chr>
1 Agnostic

```

```

2 Atheist
3 Buddhist
4 Catholic
5 DoNotKnowOrRefused
6 EvangelicalProt
7 Hindu
8 HistoricallyBlackProt
9 JehovahsWitness
10 Jewish
11 MainlineProt
12 Mormon
13 Muslim
14 Orthodox
15 OtherChristian
16 OtherFaiths
17 OtherWorldReligions
18 Unaffiliated

```

Note, above that once the data frame is reduced to one dimension by subsetting to one column, it is no longer a data frame it has been *simplified* to a vector. This might not seem like a big deal; however, it can be very frustrating and potentially break your code when you expect an object to behave like a data frame and it doesn't because it's now a vector. Alternatively, once we convert `religionWide` to a `tbl_df` via the `tbl_df()` function the object remains a data frame even when subset down to one dimension (there is no automatic simplification). Converting data frames using `tbl_df()` is not required for using `ddply` but is convenient. Also, it is important to note that `tbl_df` is simply a wrapper around a data frame that provides some additional behaviors. The newly formed `tbl_df` object will still behave like a data frame (because it technically still is a data frame) but will have some added niceties (some of which are illustrated below).

### 6.5.2 Filtering data by row

Recall the `gapminder` data. These data are available in tab-separated format in `gapminder.tsv`, and can be read in using `read.delim()` (or the related `read` functions described previously). The `read.delim()` function defaults to `header = TRUE` so this doesn't need to be specified explicitly. In this section we will be working with the `gapminder` data often, so will use a short name for the data frame to save typing.

```

> u.gm <- "http://blue.for.msu.edu/FOR875/data/gapminder.tsv"
> gm <- read.delim(u.gm)
> gm <-tbl_df(gm)

```

```
> str(gapminder)

Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 6 variables:
$ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 ...
$ continent: Factor w/ 5 levels "Africa","Americas",...: 3 3 3 3 3 3 3 3 3 ...
$ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
$ lifeExp  : num 346 364 384 408 433 ...
$ pop      : int 8425333 9240934 10267083 11537966 13079460 14880372 12881816 138679...
$ gdpPercap: num 779 821 853 836 740 ...

> head(gm)

# A tibble: 6 x 6
  country     year     pop continent lifeExp gdpPercap
  <fct>     <int>   <dbl> <fct>      <dbl>      <dbl>
1 Afghanistan 1952 8425333 Asia        28.8      779.
2 Afghanistan 1957 9240934 Asia        30.3      821.
3 Afghanistan 1962 10267083 Asia        32.0      853.
4 Afghanistan 1967 11537966 Asia        34.0      836.
5 Afghanistan 1972 13079460 Asia        36.1      740.
6 Afghanistan 1977 14880372 Asia        38.4      786.
```

Filtering helps us to examine subsets of the data such as data from a particular country or from several specified countries, data from certain years, from countries with certain populations, etc. Some examples:

```
> filter(gm, country == "Brazil")

# A tibble: 12 x 6
  country     year     pop continent lifeExp gdpPercap
  <fct>     <int>   <dbl> <fct>      <dbl>      <dbl>
1 Brazil    1952 56602560 Americas   50.9      2109.
2 Brazil    1957 65551171 Americas   53.3      2487.
3 Brazil    1962 76039390 Americas   55.7      3337.
4 Brazil    1967 88049823 Americas   57.6      3430.
5 Brazil    1972 100840058 Americas   59.5      4986.
6 Brazil    1977 114313951 Americas   61.5      6660.
7 Brazil    1982 128962939 Americas   63.3      7031.
8 Brazil    1987 142938076 Americas   65.2      7807.
9 Brazil    1992 155975974 Americas   67.1      6950.
10 Brazil   1997 168546719 Americas   69.4      7958.
11 Brazil   2002 179914212 Americas   71.0      8131.
12 Brazil   2007 190010647 Americas   72.4      9066.
```

```
> filter(gm, country %in% c("Brazil", "Mexico"))

# A tibble: 24 x 6
  country   year     pop continent lifeExp gdpPercap
  <fct>   <int>   <dbl> <fct>      <dbl>      <dbl>
1 Brazil    1952 56602560 Americas    50.9     2109.
2 Brazil    1957 65551171 Americas    53.3     2487.
3 Brazil    1962 76039390 Americas    55.7     3337.
4 Brazil    1967 88049823 Americas    57.6     3430.
5 Brazil    1972 100840058 Americas    59.5     4986.
6 Brazil    1977 114313951 Americas   61.5     6660.
7 Brazil    1982 128962939 Americas   63.3     7031.
8 Brazil    1987 142938076 Americas   65.2     7807.
9 Brazil    1992 155975974 Americas   67.1     6950.
10 Brazil   1997 168546719 Americas   69.4     7958.
# ... with 14 more rows

> filter(gm, country %in% c("Brazil", "Mexico") & year %in% c(1952,
+ 1972))

# A tibble: 4 x 6
  country   year     pop continent lifeExp gdpPercap
  <fct>   <int>   <dbl> <fct>      <dbl>      <dbl>
1 Brazil    1952 56602560 Americas    50.9     2109.
2 Brazil    1972 100840058 Americas    59.5     4986.
3 Mexico   1952 30144317 Americas    50.8     3478.
4 Mexico   1972 55984294 Americas    62.4     6809.

> filter(gm, pop > 300000000)

# A tibble: 25 x 6
  country   year     pop continent lifeExp gdpPercap
  <fct>   <int>   <dbl> <fct>      <dbl>      <dbl>
1 China     1952 556263528. Asia       44     400.
2 China     1957 637408000  Asia      50.5     576.
3 China     1962 665770000  Asia      44.5     488.
4 China     1967 754550000  Asia      58.4     613.
5 China     1972 862030000  Asia      63.1     677.
6 China     1977 943455000  Asia      64.0     741.
7 China     1982 1000281000 Asia      65.5     962.
8 China     1987 1084035000 Asia      67.3    1379.
9 China     1992 1164970000 Asia      68.7    1656.
10 China    1997 1230075000 Asia      70.4    2289.
# ... with 15 more rows
```

```
> filter(gm, pop > 300000000 & year == 2007)

# A tibble: 3 x 6
  country      year    pop continent lifeExp gdpPercap
  <fct>      <int>  <dbl> <fct>     <dbl>     <dbl>
1 China        2007 1318683096 Asia      73.0     4959.
2 India        2007 1110396331 Asia      64.7     2452.
3 United States 2007 301139947 Americas  78.2     42952.
```

Notice the full results are not printed. For example, when we asked for the data for Brazil and Mexico, only the first ten rows were printed. This is an effect of using the `tbl_df()` function. Of course if we wanted to analyze the results (as we will below) the full set of data would be available.

### 6.5.3 Selecting variables by column

Continuing with the `gapminder` data, another common task is to restrict attention to some subset of variables in the data set. The `select()` function does this.

```
> dplyr::select(gm, country, year, lifeExp)
```

```
# A tibble: 1,704 x 3
  country      year lifeExp
  <fct>      <int>  <dbl>
1 Afghanistan 1952    28.8
2 Afghanistan 1957    30.3
3 Afghanistan 1962    32.0
4 Afghanistan 1967    34.0
5 Afghanistan 1972    36.1
6 Afghanistan 1977    38.4
7 Afghanistan 1982    39.9
8 Afghanistan 1987    40.8
9 Afghanistan 1992    41.7
10 Afghanistan 1997   41.8
# ... with 1,694 more rows
```

```
> select(gm, 2:4)
```

```
# A tibble: 1,704 x 3
  year    pop continent
  <int>  <dbl> <fct>
1 1952  8425333 Asia
2 1957  9240934 Asia
```

```
3 1962 10267083 Asia
4 1967 11537966 Asia
5 1972 13079460 Asia
6 1977 14880372 Asia
7 1982 12881816 Asia
8 1987 13867957 Asia
9 1992 16317921 Asia
10 1997 22227415 Asia
# ... with 1,694 more rows

> select(gm, -c(2, 3, 4))

# A tibble: 1,704 x 3
  country     lifeExp gdpPercap
  <fct>       <dbl>     <dbl>
1 Afghanistan 28.8      779.
2 Afghanistan 30.3      821.
3 Afghanistan 32.0      853.
4 Afghanistan 34.0      836.
5 Afghanistan 36.1      740.
6 Afghanistan 38.4      786.
7 Afghanistan 39.9      978.
8 Afghanistan 40.8      852.
9 Afghanistan 41.7      649.
10 Afghanistan 41.8      635.
# ... with 1,694 more rows

> select(gm, starts_with("c"))

# A tibble: 1,704 x 2
  country continent
  <fct>    <fct>
1 Afghanistan Asia
2 Afghanistan Asia
3 Afghanistan Asia
4 Afghanistan Asia
5 Afghanistan Asia
6 Afghanistan Asia
7 Afghanistan Asia
8 Afghanistan Asia
9 Afghanistan Asia
10 Afghanistan Asia
# ... with 1,694 more rows
```

Notice a few things. Variables can be selected by name or column number. As usual, a negative sign tells R to leave something out. And there are special functions such as `starts_with` that provide ways to match part of a variable's name.

### 6.5.4 Pipes

Consider selecting the country, year, and population for countries in Asia or Europe. One possibility is to nest a `filter()` function inside a `select()` function.

```
> select(filter(gm, continent %in% c("Asia", "Europe")), country,
+       year, pop)

# A tibble: 756 x 3
  country     year     pop
  <fct>     <int>   <dbl>
  1 Afghanistan 1952 8425333
  2 Afghanistan 1957 9240934
  3 Afghanistan 1962 10267083
  4 Afghanistan 1967 11537966
  5 Afghanistan 1972 13079460
  6 Afghanistan 1977 14880372
  7 Afghanistan 1982 12881816
  8 Afghanistan 1987 13867957
  9 Afghanistan 1992 16317921
 10 Afghanistan 1997 22227415
# ... with 746 more rows
```

Even a two-step process like this becomes hard to follow in this nested form, and often we will want to perform more than two operations. There is a nice feature in `dplyr` that allows us to “feed” results of one function into the first argument of a subsequent function. Another way of saying this is that we are “piping” the results into another function. The `%>%` operator does the piping. Here we again restrict attention to country, year, and population for countries in Asia or Europe.

```
> gm %>% filter(continent %in% c("Asia", "Europe")) %>% select(country,
+       year, pop)

# A tibble: 756 x 3
  country     year     pop
  <fct>     <int>   <dbl>
  1 Afghanistan 1952 8425333
  2 Afghanistan 1957 9240934
  3 Afghanistan 1962 10267083
```

```

4 Afghanistan 1967 11537966
5 Afghanistan 1972 13079460
6 Afghanistan 1977 14880372
7 Afghanistan 1982 12881816
8 Afghanistan 1987 13867957
9 Afghanistan 1992 16317921
10 Afghanistan 1997 22227415
# ... with 746 more rows

```

It can help to think of `%>%` as representing the word “then.” The above can be read as, “Start with the data frame `gm` *then* filter it to select data from the continents Asia and Europe *then* select the variables country, year, and population from these data.”

The pipe operator `%>%` is not restricted to functions in `dplyr`. In fact the pipe operator itself was introduced in another package called `magrittr`, but is included in `dplyr` as a convenience.

### 6.5.5 Arranging data by row

By default the `gapminder` data are arranged by country and then by year.

```

> head(gm, 15)

# A tibble: 15 x 6
  country     year   pop continent lifeExp gdpPercap
  <fct>     <int> <dbl> <fct>      <dbl>    <dbl>
1 Afghanistan 1952  8425333 Asia        28.8    779.
2 Afghanistan 1957  9240934 Asia        30.3    821.
3 Afghanistan 1962 10267083 Asia        32.0    853.
4 Afghanistan 1967 11537966 Asia        34.0    836.
5 Afghanistan 1972 13079460 Asia        36.1    740.
6 Afghanistan 1977 14880372 Asia        38.4    786.
7 Afghanistan 1982 12881816 Asia        39.9    978.
8 Afghanistan 1987 13867957 Asia        40.8    852.
9 Afghanistan 1992 16317921 Asia        41.7    649.
10 Afghanistan 1997 22227415 Asia        41.8    635.
11 Afghanistan 2002 25268405 Asia        42.1    727.
12 Afghanistan 2007 31889923 Asia        43.8    975.
13 Albania     1952 1282697 Europe      55.2   1601.
14 Albania     1957 1476505 Europe      59.3   1942.
15 Albania     1962 1728137 Europe      64.8   2313.

```

Possibly arranging the data by year and then country would be desired. The `arrange()` function makes this easy. We will again use pipes.

```
> gm %>% arrange(year, country)

# A tibble: 1,704 x 6
  country      year    pop continent lifeExp gdpPercap
  <fct>      <int>  <dbl> <fct>      <dbl>     <dbl>
1 Afghanistan 1952 8425333 Asia        28.8     779.
2 Albania      1952 1282697 Europe     55.2    1601.
3 Algeria       1952 9279525 Africa     43.1    2449.
4 Angola        1952 4232095 Africa     30.0    3521.
5 Argentina     1952 17876956 Americas   62.5    5911.
6 Australia     1952 8691212 Oceania    69.1   10040.
7 Austria       1952 6927772 Europe     66.8    6137.
8 Bahrain        1952 120447 Asia       50.9    9867.
9 Bangladesh     1952 46886859 Asia       37.5    684.
10 Belgium       1952 8730405 Europe     68      8343.
# ... with 1,694 more rows
```

How about the data for Rwanda, arranged in order of life expectancy.

```
> gm %>% filter(country == "Rwanda") %>% arrange(lifeExp)

# A tibble: 12 x 6
  country    year    pop continent lifeExp gdpPercap
  <fct>    <int>  <dbl> <fct>      <dbl>     <dbl>
1 Rwanda    1992 7290203 Africa     23.6     737.
2 Rwanda    1997 7212583 Africa     36.1     590.
3 Rwanda    1952 2534927 Africa     40       493.
4 Rwanda    1957 2822082 Africa     41.5     540.
5 Rwanda    1962 3051242 Africa     43       597.
6 Rwanda    2002 7852401 Africa     43.4     786.
7 Rwanda    1987 6349365 Africa     44.0     848.
8 Rwanda    1967 3451079 Africa     44.1     511.
9 Rwanda    1972 3992121 Africa     44.6     591.
10 Rwanda   1977 4657072 Africa     45       670.
11 Rwanda   1982 5507565 Africa     46.2     882.
12 Rwanda   2007 8860588 Africa     46.2     863.
```

Possibly we want these data to be in decreasing (descending) order. Here, `desc()` is one of many `dplyr` helper functions.

```
> gm %>% filter(country == "Rwanda") %>% arrange(desc(lifeExp))

# A tibble: 12 x 6
```

|    | country | year  | pop     | continent | lifeExp | gdpPerCap |
|----|---------|-------|---------|-----------|---------|-----------|
|    | <fct>   | <int> | <dbl>   | <fct>     | <dbl>   | <dbl>     |
| 1  | Rwanda  | 2007  | 8860588 | Africa    | 46.2    | 863.      |
| 2  | Rwanda  | 1982  | 5507565 | Africa    | 46.2    | 882.      |
| 3  | Rwanda  | 1977  | 4657072 | Africa    | 45      | 670.      |
| 4  | Rwanda  | 1972  | 3992121 | Africa    | 44.6    | 591.      |
| 5  | Rwanda  | 1967  | 3451079 | Africa    | 44.1    | 511.      |
| 6  | Rwanda  | 1987  | 6349365 | Africa    | 44.0    | 848.      |
| 7  | Rwanda  | 2002  | 7852401 | Africa    | 43.4    | 786.      |
| 8  | Rwanda  | 1962  | 3051242 | Africa    | 43      | 597.      |
| 9  | Rwanda  | 1957  | 2822082 | Africa    | 41.5    | 540.      |
| 10 | Rwanda  | 1952  | 2534927 | Africa    | 40      | 493.      |
| 11 | Rwanda  | 1997  | 7212583 | Africa    | 36.1    | 590.      |
| 12 | Rwanda  | 1992  | 7290203 | Africa    | 23.6    | 737.      |

Possibly we want to include only the year and life expectancy, to make the message more stark.

```
> gm %>% filter(country == "Rwanda") %>% select(year, lifeExp) %>%
+     arrange(desc(lifeExp))

# A tibble: 12 x 2
  year lifeExp
  <int>   <dbl>
1 2007    46.2
2 1982    46.2
3 1977    45
4 1972    44.6
5 1967    44.1
6 1987    44.0
7 2002    43.4
8 1962    43
9 1957    41.5
10 1952    40
11 1997    36.1
12 1992    23.6
```

For analyzing data in R, the order shouldn't matter. But for presentation to human eyes, the order is important.

### 6.5.6 Renaming variables

The `dplyr` package has a `rename` function that makes renaming variables in a data frame quite easy.

```
> gm <- rename(gm, population = pop)
> head(gm)

# A tibble: 6 x 6
  country      year population continent lifeExp gdpPercap
  <fct>     <int>    <dbl> <fct>      <dbl>    <dbl>
1 Afghanistan 1952     8425333 Asia       28.8     779.
2 Afghanistan 1957     9240934 Asia       30.3     821.
3 Afghanistan 1962    10267083 Asia       32.0     853.
4 Afghanistan 1967    11537966 Asia       34.0     836.
5 Afghanistan 1972    13079460 Asia       36.1     740.
6 Afghanistan 1977    14880372 Asia       38.4     786.
```

### 6.5.7 Data summaries and grouping

The `summarize()` function computes summary statistics or user provided function for one or more columns of data in a data frame.

```
> summarize(gm, meanpop = mean(population), medpop = median(population))

# A tibble: 1 x 2
  meanpop   medpop
  <dbl>     <dbl>
1 29601212. 7023596.

> ## or
> gm %>% summarize(meanpop = mean(population), medpop = median(population))

# A tibble: 1 x 2
  meanpop   medpop
  <dbl>     <dbl>
1 29601212. 7023596.
```

Often we want summaries for specific components of the data. For example, we might want the median life expectancy for each continent separately. One option is subsetting:

```
> median(gm$lifeExp[gm$continent == "Africa"])

[1] 47.792

> median(gm$lifeExp[gm$continent == "Asia"])
```

```
[1] 61.7915

> median(gm$lifeExp[gm$continent == "Europe"])

[1] 72.241

> median(gm$lifeExp[gm$continent == "Americas"])

[1] 67.048

> median(gm$lifeExp[gm$continent == "Oceania"])

[1] 73.665
```

The `group_by()` function makes this easier, and makes the output more useful.

```
> gm %>% group_by(continent) %>% summarize(medLifeExp = median(lifeExp))

# A tibble: 5 x 2
  continent medLifeExp
  <fct>        <dbl>
1 Africa        47.8
2 Americas      67.0
3 Asia          61.8
4 Europe        72.2
5 Oceania       73.7
```

Or if we want the results ordered by the median life expectancy:

```
> gm %>% group_by(continent) %>%
+   summarize(medLifeExp = median(lifeExp)) %>%
+   arrange(medLifeExp)

# A tibble: 5 x 2
  continent medLifeExp
  <fct>        <dbl>
1 Africa        47.8
2 Asia          61.8
3 Americas      67.0
4 Europe        72.2
5 Oceania       73.7
```

As another example, we calculate the number of observations we have per continent (using the `n()` helper function), and then, among continents, how many distinct countries are represented (using `n_distinct()`).

```
> gm %>% group_by(continent) %>% summarize(numObs = n())

# A tibble: 5 x 2
  continent numObs
  <fct>     <int>
1 Africa      624
2 Americas    300
3 Asia        396
4 Europe      360
5 Oceania     24

> gm %>%
+   group_by(continent) %>%
+   summarize(n_obs = n(), n_countries = n_distinct(country))

# A tibble: 5 x 3
  continent n_obs n_countries
  <fct>     <int>       <int>
1 Africa      624          52
2 Americas    300          25
3 Asia        396          33
4 Europe      360          30
5 Oceania     24           2
```

Here is a bit more involved example that calculates the minimum and maximum life expectancies for countries in Africa by year.

```
> gm %>%
+   filter(continent == "Africa") %>%
+   group_by(year) %>%
+   summarize(min_lifeExp = min(lifeExp), max_lifeExp = max(lifeExp))

# A tibble: 12 x 3
  year min_lifeExp max_lifeExp
  <int>      <dbl>      <dbl>
1 1952         30       52.7
2 1957        31.6      58.1
3 1962        32.8      60.2
4 1967        34.1      61.6
5 1972        35.4      64.3
```

|    |      |      |      |
|----|------|------|------|
| 6  | 1977 | 36.8 | 67.1 |
| 7  | 1982 | 38.4 | 69.9 |
| 8  | 1987 | 39.9 | 71.9 |
| 9  | 1992 | 23.6 | 73.6 |
| 10 | 1997 | 36.1 | 74.8 |
| 11 | 2002 | 39.2 | 75.7 |
| 12 | 2007 | 39.6 | 76.4 |

This is interesting, but the results don't include the countries that achieved the minimum and maximum life expectancies. Here is one way to achieve that. We will start with the minimum life expectancy. Note the rank of the minimum value will be 1.

```
> gm %>% select(country, continent, year, lifeExp) %>%
+   group_by(year) %>% arrange(year) %>%
+   filter(rank(lifeExp) == 1)

# A tibble: 12 x 4
# Groups:   year [12]
  country     continent   year lifeExp
  <fct>       <fct>      <int>   <dbl>
1 Afghanistan Asia        1952    28.8
2 Afghanistan Asia        1957    30.3
3 Afghanistan Asia        1962    32.0
4 Afghanistan Asia        1967    34.0
5 Sierra Leone Africa    1972    35.4
6 Cambodia      Asia        1977    31.2
7 Sierra Leone Africa    1982    38.4
8 Angola        Africa     1987    39.9
9 Rwanda         Africa     1992    23.6
10 Rwanda        Africa    1997    36.1
11 Zambia        Africa    2002    39.2
12 Swaziland     Africa    2007    39.6
```

Next we add the maximum life expectancy. Here we need to better understand the `desc()` function, which will transform a vector into a numeric vector which will be sorted in descending order. Here are some examples.

```
> desc(1:5)
[1] -1 -2 -3 -4 -5

> desc(c(2, 3, 1, 5, 6, -4))
```

```
[1] -2 -3 -1 -5 -6  4

> desc(c("a", "c", "b", "w", "e"))

[1] -1 -3 -2 -5 -4
```

We now use this to extract the maximum life expectancy. Recall that `|` represents “or.” Also by default only the first few rows of a `tbl_df` object will be printed. To see all the rows we pipe the output to `print(n = 24)` to ask for all 24 rows to be printed.

```
> gm %>% select(country, continent, year, lifeExp) %>%
+   group_by(year) %>% arrange(year) %>%
+   filter(rank(lifeExp) == 1 | rank(desc(lifeExp)) == 1) %>%
+   print(n=24)

# A tibble: 24 x 4
# Groups:   year [12]
  country     continent   year lifeExp
  <fct>       <fct>     <int>   <dbl>
1 Afghanistan Asia      1952    28.8
2 Norway      Europe    1952    72.7
3 Afghanistan Asia      1957    30.3
4 Iceland     Europe    1957    73.5
5 Afghanistan Asia      1962    32.0
6 Iceland     Europe    1962    73.7
7 Afghanistan Asia      1967    34.0
8 Sweden      Europe    1967    74.2
9 Sierra Leone Africa   1972    35.4
10 Sweden     Europe    1972    74.7
11 Cambodia   Asia      1977    31.2
12 Iceland    Europe    1977    76.1
13 Japan       Asia      1982    77.1
14 Sierra Leone Africa   1982    38.4
15 Angola     Africa    1987    39.9
16 Japan       Asia      1987    78.7
17 Japan       Asia      1992    79.4
18 Rwanda     Africa    1992    23.6
19 Japan       Asia      1997    80.7
20 Rwanda     Africa    1997    36.1
21 Japan       Asia      2002    82
22 Zambia     Africa    2002    39.2
23 Japan       Asia      2007    82.6
24 Swaziland  Africa    2007    39.6
```

### 6.5.8 Creating new variables

The \$ notation provides a simple way to create new variables in a data frame. The `mutate()` function provides another, sometimes cleaner way to do this. We will use `mutate()` along with the `lag()` function to investigate changes in life expectancy over five years for the `gapminder` data. We'll do this in a few steps. First, we create a variable that measures the change in life expectancy and remove the population and GDP variables that are not of interest. We have to be careful to first group by country, since we want to calculate the change in life expectancy by country.

```
> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% select(-c(population,
+   gdpPercap))

# A tibble: 1,704 x 5
# Groups:   country [142]
  country     year continent lifeExp changeLifeExp
  <fct>     <int> <fct>     <dbl>      <dbl>
1 Afghanistan 1952 Asia       28.8        NA
2 Afghanistan 1957 Asia       30.3       1.53
3 Afghanistan 1962 Asia       32.0       1.66
4 Afghanistan 1967 Asia       34.0       2.02
5 Afghanistan 1972 Asia       36.1       2.07
6 Afghanistan 1977 Asia       38.4       2.35
7 Afghanistan 1982 Asia       39.9       1.42
8 Afghanistan 1987 Asia       40.8      0.968
9 Afghanistan 1992 Asia       41.7      0.852
10 Afghanistan 1997 Asia      41.8      0.0890
# ... with 1,694 more rows
```

Next, summarize by computing the largest drop in life expectancy.

```
> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% select(-c(population,
+   gdpPercap)) %>% summarize(largestDropLifeExp = min(changeLifeExp))

# A tibble: 142 x 2
  country     largestDropLifeExp
  <fct>           <dbl>
1 Afghanistan        NA
2 Albania            NA
3 Algeria            NA
4 Angola             NA
5 Argentina          NA
```

```

6 Australia          NA
7 Austria           NA
8 Bahrain            NA
9 Bangladesh         NA
10 Belgium           NA
# ... with 132 more rows

```

Oops. We forgot that since we don't have data from before 1952, the first drop will be NA. Let's try again.

```

> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% select(-c(population,
+   gdpPercap)) %>% summarize(largestDropLifeExp = min(changeLifeExp,
+   na.rm = TRUE))

# A tibble: 142 x 2
  country      largestDropLifeExp
  <fct>          <dbl>
1 Afghanistan    0.0890
2 Albania        -0.419
3 Algeria         1.31
4 Angola          -0.036
5 Argentina       0.492
6 Australia        0.170
7 Austria          0.490
8 Bahrain          0.84
9 Bangladesh       1.67
10 Belgium         0.5
# ... with 132 more rows

```

That's not quite what we wanted. We could arrange the results by the life expectancy drop, but it would be good to have both the continent and year printed out also. So we'll take a slightly different approach, by arranging the results in increasing order.

```

> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% select(-c(population,
+   gdpPercap)) %>% arrange(changeLifeExp)

# A tibble: 1,704 x 5
# Groups:   country [142]
  country      year continent lifeExp changeLifeExp
  <fct>      <int> <fct>     <dbl>        <dbl>
1 Afghanistan  1952 Europe      67.0        -0.0890
2 Albania      1952 Europe      67.0        -0.419
3 Algeria      1952 Europe      67.0         1.31
4 Angola       1952 Europe      67.0        -0.036
5 Argentina    1952 Europe      67.0         0.492
6 Australia    1952 Europe      67.0         0.170
7 Austria      1952 Europe      67.0         0.490
8 Bahrain      1952 Europe      67.0         0.84
9 Bangladesh   1952 Europe      67.0         1.67
10 Belgium     1952 Europe      67.0         0.5
# ... with 132 more rows

```

```

1 Rwanda      1992 Africa     23.6    -20.4
2 Zimbabwe   1997 Africa     46.8    -13.6
3 Lesotho     2002 Africa     44.6    -11.0
4 Swaziland   2002 Africa     43.9    -10.4
5 Botswana    1997 Africa     52.6    -10.2
6 Cambodia    1977 Asia       31.2    -9.10
7 Namibia     2002 Africa     51.5    -7.43
8 South Africa 2002 Africa     53.4    -6.87
9 Zimbabwe    2002 Africa     40.0    -6.82
10 China      1962 Asia       44.5    -6.05
# ... with 1,694 more rows

```

That's still not quite right. Because the data are grouped by country, R did the ordering within group. If we want to see the largest drops overall, we need to remove the grouping.

```

> gm %>% group_by(country) %>% mutate(changeLifeExp = lifeExp -
+   lag(lifeExp, order_by = year)) %>% select(-c(population,
+   gdpPercap)) %>% ungroup() %>% arrange(changeLifeExp) %>%
+   print(n = 20)

# A tibble: 1,704 x 5
  country      year continent lifeExp changeLifeExp
  <fct>        <int> <fct>     <dbl>      <dbl>
1 Rwanda      1992 Africa     23.6    -20.4
2 Zimbabwe   1997 Africa     46.8    -13.6
3 Lesotho     2002 Africa     44.6    -11.0
4 Swaziland   2002 Africa     43.9    -10.4
5 Botswana    1997 Africa     52.6    -10.2
6 Cambodia    1977 Asia       31.2    -9.10
7 Namibia     2002 Africa     51.5    -7.43
8 South Africa 2002 Africa     53.4    -6.87
9 Zimbabwe    2002 Africa     40.0    -6.82
10 China      1962 Asia       44.5    -6.05
11 Botswana   2002 Africa     46.6    -5.92
12 Zambia      1997 Africa     40.2    -5.86
13 Iraq         1992 Asia      59.5    -5.58
14 Liberia     1992 Africa     40.8    -5.23
15 Cambodia    1972 Asia       40.3    -5.10
16 Kenya        1997 Africa     54.4    -4.88
17 Somalia      1992 Africa     39.7    -4.84
18 Zambia      1992 Africa     46.1    -4.72
19 Swaziland   2007 Africa     39.6    -4.26

```

```
20 Uganda      1997 Africa      44.6      -4.25
# ... with 1,684 more rows
```

**Exercise 7**

Learning objectives: introduce `with`, `tapply`, and `cut` functions; summarize data using the `table` function with logical subsetting; practice using factor data types.

**Exercise 8**

Learning objectives: work with messy data; import data from an external spreadsheet; practice using functions in `tidyverse` and `ggplot2`.

**Exercise 9**

Learning objectives: work with several key `dplyr` functions; manipulate data frames (actually tibbles); summarize and visualize data from large data files.

# Chapter 7

## Functions and Programming

We have been working with a wide variety of R functions, from simple functions such as `mean()` and `sd()` to more complex functions such as `ggplot()` and `apply()`. Gaining a better understanding of existing functions and ability to write your own functions dramatically increases what we can do with R. Learning about R's programming capabilities is an important step in gaining facility with functions.

### 7.1 R functions

Data on the yield, in pounds per acre, of two types of corn seeds (regular and kiln dried) were collected. Each of the 11 plots of land was split into two subplots, and one of the subplots was planted in regular corn while the other was planted in kiln dried corn. These data were analyzed in a famous paper of William Gosset. Here are the data.

```
> u.corn <- "http://blue.for.msu.edu/FOR875/data/corn.csv"
> corn <- read.csv(u.corn, header = TRUE)
> corn

  regular kiln_dried
1     1903      2009
2     1935      1915
3     1910      2011
4     2496      2463
5     2108      2180
6     1961      1925
7     2060      2122
8     1444      1482
9     1612      1542
10    1316      1443
11    1511      1535
```

A paired t test, or a confidence interval for the mean difference, may be used to assess the difference in yield between the two varieties. Of course R has a function `t.test` that performs a paired t test and compute a confidence interval, but we will perform the test without using that function. We will focus for now on testing the hypotheses  $H_0: \mu_d = 0$  versus  $H_a: \mu_d \neq 0$  and on a two-sided confidence interval for  $\mu_d$ . Here  $\mu_d$  represents the population mean difference.

The paired  $t$  statistic is defined by

$$t = \frac{\bar{d}}{S_d/\sqrt{n}} \quad (7.1)$$

where  $\bar{d}$  is the mean of the differences,  $S_d$  is the standard deviation of the differences, and  $n$  is the sample size. The p-value is twice the area to the right of  $|t_{\text{obs}}|$ , where  $t_{\text{obs}}$  is the observed  $t$  statistic, and a confidence interval is given by

$$\bar{d} \pm t^*(S_d/\sqrt{n}). \quad (7.2)$$

Here  $t^*$  is an appropriate quantile of a  $t$  distribution with  $n - 1$  degrees of freedom.

```
> dbar <- mean(corn$kiln_dried - corn$regular)
> n <- length(corn$regular)
> S_d <- sd(corn$kiln_dried - corn$regular)
> t_obs <- dbar/(S_d/sqrt(n))
> t_obs

[1] 1.690476

> pval <- 2 * (1 - pt(abs(t_obs), n - 1))
> pval

[1] 0.1218166

> margin <- qt(0.975, n - 1) * (S_d/sqrt(n))
> lcl <- dbar - margin
> ucl <- dbar + margin
> lcl

[1] -10.7271

> ucl

[1] 78.18164
```

With a few lines of R code we have calculated the  $t$  statistic, the p-value, and the confidence interval. Since paired t tests are pretty common, however, it would be

helpful to automate this procedure. One obvious reason is to save time and effort, but another important reason is to avoid mistakes. It would be easy to make a mistake (e.g., using  $n$  instead of  $n - 1$  as the degrees of freedom) when repeating the above computations.

Here is a first basic function which automates the computation.

```
> paired_t <- function(x1, x2) {
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   margin <- qt(0.975, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
```

And here is the function in action

```
> paired_t(x1 = corn$kiln_dried, x2 = corn$regular)

$tstat
[1] 1.690476

$pval
[1] 0.1218166

$lcl
[1] -10.7271

$ucl
[1] 78.18164
```

An explanation and comments on the function are in order.

- `paired_t <- function(x1, x2)` assigns a function of two variables, `x1` and `x2`, to an R object called `paired_t`.
- The *compound expression*, i.e., the code that makes up the body of the function, is enclosed in curly braces `{}`.
- `return(list(tstat = tstat, pval = pval, lcl=lcl, ucl=ucl))` indicates the object(s) returned by the function. In this case the function returns a list with three components.

- The body of the function basically mimics the computations required to compute the t statistic, the p-value, and the confidence interval.
- Several objects such as `n` and `dbar` are created in the body of the function. These objects are NOT available outside the function. We will discuss this further when we cover environments and scope in R.

Our function has automated the basic calculations. But it is still somewhat limited in usefulness. For example, it only computes a 95% confidence interval, while a user may want a different confidence level. And the function only performs a two-sided test, while a user may want a one-sided procedure. We modify the function slightly to allow the user to specify the confidence level next.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
```

```
> paired_t(corn$kiln_dried, corn$regular)

$tstat
[1] 1.690476

$pval
[1] 0.1218166

$lcl
[1] -10.7271

$ucl
[1] 78.18164

> paired_t(corn$kiln_dried, corn$regular, cl = 0.99)

$tstat
```

```
[1] 1.690476
```

```
$pval
[1] 0.1218166
```

```
$lcl
[1] -29.50398
```

```
$ucl
[1] 96.95853
```

Two things to note. First, arguments do not have to be named. So

```
paired_t(corn$kiln_dried, corn$regular)
```

and

```
paired_t(x1 = corn$kiln_dried, x2 = corn$regular)
```

are equivalent. But we need to be careful if we do not name arguments because then we have to know the ordering of the arguments in the function declaration.

Second, in the declaration of the function, the third argument `c1` was given a default value of 0.95. If a user does not specify a value for `c1` it will silently be set to 0.95. But of course a user can override this, as we did in

```
paired_t(corn$kiln_dried, corn$regular, c1 = 0.99)
```

### 7.1.1 Creating functions

Creating very short functions at the command prompt is a reasonable strategy. For longer functions, one option is to write the function in a script and then submit the whole function. Or a function can be written in any text editor, saved as a plain text file (possibly with a `.r` extension), and then read into R using the `source()` function.

## 7.2 Programming: Conditional Statements

The `paired_t` function is somewhat useful, but could be improved in several ways. For example, consider the following:

```
> paired_t(1:5, 1:4)
```

```
Warning in x1 - x2: longer object length is not a multiple of shorter object
length
```

```
Warning in x1 - x2: longer object length is not a multiple of shorter object
length
```

```
$tstat
[1] 1

$pval
[1] 0.373901

$lcl
[1] -1.421156

$ucl
[1] 3.021156
```

The user specified data that had different numbers of observations in `x1` and `x2`, which of course can't be data tested by a paired t test. Rather than stopping and letting the user know that this is a problem, the function continued and produced meaningless output.

Also, the function as written only allows testing against a two-sided alternative hypothesis, and it would be good to allow one-sided alternatives.

First we will address some checks on arguments specified by the user. For this we will use an `if()` function and a `stop()` function.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors must have the same length")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
> paired_t(1:5, 1:4)
```

```
Error in paired_t(1:5, 1:4): The input vectors must have the same length
```

The argument to the `if()` function is evaluated. If the argument returns `TRUE` the ensuing code is executed. Otherwise, the ensuing code is skipped and the rest of the function is evaluated. If a `stop()` function is executed, the function is exited and the argument of `stop()` is printed.

To better understand and use `if()` statements, we need to understand comparison operators and logical operators.

### 7.2.1 Comparison and logical operators

We have made use of some of the comparison operators in R. These include

- Equal: `==`
- Not equal: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Special care needs to be taken with the `==` and `!=` operators because of how numbers are represented on computers, see Section 7.3.

There are also three logical operators, with two variants of the “and” operator and the “or” operator.

- and: Either `&` or `&&`
- or: Either `|` or `||`
- not: `!`

The “double” operators `&&` and `||` just examine the first element of the two vectors, whereas the “single” operators `&` and `|` compare element by element.

```
> c(FALSE, TRUE, FALSE) || c(TRUE, FALSE, FALSE)
```

```
[1] TRUE
```

```
> c(FALSE, TRUE, FALSE) | c(TRUE, FALSE, FALSE)
```

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

```
> c(FALSE, TRUE, FALSE) && c(TRUE, TRUE, FALSE)
```

```
[1] FALSE

> c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)

[1] FALSE TRUE FALSE
```

We can use the logical operators to check whether a user-specified confidence level is between 0 and 1.

```
> paired_t <- function(x1, x2, cl = 0.95) {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors must have the same length")
+   }
+   if (cl <= 0 || cl >= 1) {
+     stop("The confidence level must be between 0 and 1")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   pval <- 2 * (1 - pt(abs(tstat), n - 1))
+   pctile <- 1 - (1 - cl)/2
+   margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+   lcl <- dbar - margin
+   ucl <- dbar + margin
+   return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+ }
> paired_t(1:5, 2:6, cl = 15)

Error in paired_t(1:5, 2:6, cl = 15): The confidence level must be between
0 and 1
```

### 7.2.2 If else statements

The `if()` statement we have used so far has the form

```
if (condition) {expression}
```

Often we want to evaluate one expression if the condition is true, and evaluate a different expression if the condition is false. That is accomplished by the `if else` statement. Here we determine whether a number is positive, negative, or zero.

```
> Sign <- function(x) {
+   if (x < 0) {
+     print("the number is negative")
+   } else if (x > 0) {
+     print("the number is positive")
+   } else {
+     print("the number is zero")
+   }
+ }
> Sign(3)
```

```
[1] "the number is positive"
```

```
> Sign(-3)
```

```
[1] "the number is negative"
```

```
> Sign(0)
```

```
[1] "the number is zero"
```

Notice the “different expression” for the first `if` statement was itself an `if` statement.

Next we modify the `paired_t` function to allow two-sided and one-sided alternatives.

```
> paired_t <- function(x1, x2, cl = 0.95, alternative = "not.equal") {
+   if (length(x1) != length(x2)) {
+     stop("The input vectors must be of the same length")
+   }
+   if (cl <= 0 || cl >= 1) {
+     stop("The confidence level must be between 0 and 1")
+   }
+   n <- length(x1)
+   dbar <- mean(x1 - x2)
+   s_d <- sd(x1 - x2)
+   tstat <- dbar/(s_d/sqrt(n))
+   if (alternative == "not.equal") {
+     pval <- 2 * (1 - pt(abs(tstat), n - 1))
+     pctile <- 1 - (1 - cl)/2
+     margin <- qt(pctile, n - 1) * s_d/sqrt(n)
+     lcl <- dbar - margin
+     ucl <- dbar + margin
```

```
+     } else if (alternative == "greater") {
+       pval <- 1 - pt(tstat, n - 1)
+       margin <- qt(cl, n - 1) * s_d/sqrt(n)
+       lcl <- dbar - margin
+       ucl <- Inf
+     } else if (alternative == "less") {
+       pval <- pt(tstat, n - 1)
+       margin <- qt(cl, n - 1) * s_d/sqrt(n)
+       lcl <- -Inf
+       ucl <- dbar + margin
+     }
+
+     return(list(tstat = tstat, pval = pval, lcl = lcl, ucl = ucl))
+   }
> paired_t(corn$kiln_dried, corn$regular)

$tstat
[1] 1.690476

$pval
[1] 0.1218166

$lcl
[1] -10.7271

$ucl
[1] 78.18164

> paired_t(corn$kiln_dried, corn$regular, alternative = "less")

$tstat
[1] 1.690476

$pval
[1] 0.9390917

$lcl
[1] -Inf

$ucl
[1] 69.88831

> paired_t(corn$kiln_dried, corn$regular, alternative = "greater")
```

```
$tstat  
[1] 1.690476  
  
$pval  
[1] 0.0609083  
  
$lcl  
[1] -2.433766  
  
$ucl  
[1] Inf
```

## 7.3 Computer Arithmetic

Like most software, R does not perform exact arithmetic. Rather, R follows the IEEE 754 floating point standards. This can have profound effects on how computational algorithms are implemented, but is also important when considering things like comparisons.

Note first that computer arithmetic does not follow some of the rules of ordinary arithmetic. For example, it is not associative.

```
> 2^-30  
  
[1] 0.0000000009313226  
  
> 2^-30 + (2^30 - 2^30)  
  
[1] 0.0000000009313226  
  
> (2^-30 + 2^30) - 2^30  
  
[1] 0
```

Computer arithmetic is not exact.

```
> 1.5 - 1.4  
  
[1] 0.1  
  
> 1.5 - 1.4 == 0.1
```

```
[1] FALSE

> (1.5 - 1.4) - 0.1

[1] 0.0000000000000008326673
```

So for example an `if` statement that uses an equality test may not give the expected answer. One way to avoid this problem is to test “near equality” using `all.equal()`. The function takes as arguments two objects to be compared, and a tolerance. If the objects are within the tolerance of each other, the function returns `TRUE`. The tolerance has a default value of about  $1.5 \times 10^{-8}$ , which works well in many cases.

```
> all.equal((1.5 - 1.4), 0.1)

[1] TRUE
```

## 7.4 Loops

Loops are an important component of any programming language, including R. Vectorized calculations and functions such as `apply()` make loops a bit less central to R than to many other languages, but an understanding of the three looping structures in R is still quite important.

We will investigate loops in the context of computing what is sometimes called the “machine epsilon.” Because of the inexact representation of numbers in R (and other languages) sometimes R cannot distinguish between the numbers `1` and `1 + x` for small values of `x`. The smallest value of `x` such that `1` and `1+x` are not declared equal is the machine epsilon.

```
> 1 == 1 + 10^-4

[1] FALSE

> 1 == 1 + 10^-50

[1] TRUE
```

Clearly the machine epsilon is somewhere between  $10^{-4}$  and  $10^{-50}$ . How can we find its value exactly? Since floating point numbers use a binary representation, we know that the machine epsilon will be equal to  $1/2^k$  for some value of  $k$ . So to find the machine epsilon, we can keep testing whether `1` and `1 + 1/2k` are equal, until we find a value  $k$  where the two are equal. Then the machine epsilon will be  $1/2^{k-1}$ , since it is the smallest value for which the two are NOT equal.

```
> 1 == 1 + 1/2
[1] FALSE

> 1 == 1 + 1/2^2
[1] FALSE

> 1 == 1 + 1/2^3
[1] FALSE
```

Testing by hand like this gets tedious quickly. A loop can automate the process. We will do this with the two R loop types, `repeat` and `while`.

### 7.4.1 A repeat loop

A `repeat` loop just repeats a given expression over and over again until a `break` statement is encountered.

```
> k <- 1
> repeat {
+   if (1 == 1 + 1/2^k) {
+     break
+   } else {
+     k <- k + 1
+   }
+ }
> k

[1] 53

> 1/2^(k - 1)
[1] 0.000000000000002220446
```

This code initializes `k` at 1. The body of the loop initially checks whether 1 and  $1 + 1/2^k$  are equal. If they are equal, the `break` statement is executed and control is transferred outside the loop. If they are not equal, `k` is increased by 1, and we return to the beginning of the top of the body of the loop.

### 7.4.2 A while loop

A `while` loop has the form

```
while (condition) expression
```

As long as the `condition` is TRUE the `expression` is evaluated. Once the `condition` is FALSE control is transferred outside the loop.

```
> k <- 1
> while (1 != 1 + 1/2^k) {
+     k <- k + 1
+ }
> k

[1] 53

> 1/2^(k - 1)

[1] 0.000000000000002220446
```

### 7.4.3 A for loop

A `for` loop has the form

```
for (variable in vector) expression
```

The `for` loop sets the `variable` equal to each element of the `vector` in succession, and evaluates the `expression` each time. Here are two different ways to use a `for` loop to calculate the sum of the elements in a vector.

```
> x <- 1:10
> S <- 0
> for (i in 1:length(x)) {
+     S <- S + x[i]
+ }
> S

[1] 55

> S <- 0
> for (value in x) {
+     S <- S + value
+ }
> S
```

```
[1] 55
```

In the first case we loop over the positions of the vector elements, while in the second case we loop over the elements themselves.

## 7.5 Efficiency considerations

In many contexts R and modern computers are fast enough that the user does not need to worry about writing efficient code. There are a few simple ways to write efficient code that are easy enough, and provide enough speed-up, that they are worth following as often as possible. The R function `system.time()` reports how long a set of R code takes to execute, and we will use this function to compare different ways to accomplish objectives in R.

### 7.5.1 Growing objects

Consider two ways to create a sequence of integers from 1 to n, implemented in functions `f1` and `f2`.

1. Start with a zero-length vector, and let it grow:

```
> f1 <- function(n) {
+   x <- numeric(0)
+   for (i in 1:n) {
+     x <- c(x, i)
+   }
+   x
+ }
```

2. Start with a vector of length n and fill in the values:

```
> f2 <- function(n) {
+   x <- numeric(n)
+   for (i in 1:n) {
+     x[i] <- i
+   }
+   x
+ }
```

Here are the two functions in action, with  $n = 100000$ .

```
> n <- 100000
> system.time(f1(n))

  user  system elapsed
19.060   0.004 19.077

> system.time(f2(n))

  user  system elapsed
0.100   0.000   0.098
```

It is *much* more efficient to start with a full-length vector and then fill in values.<sup>1</sup> Of course another way to create a vector of integers from 1 to  $n$  is to use `1:n`. Let's see how fast this is.

```
> system.time(1:n)

  user  system elapsed
0       0       0

> n <- 1000000
> system.time(1:n)

  user  system elapsed
0       0       0
```

For  $n = 100000$  this is so fast the system time is very close to zero. Even when  $n$  is 1000000 the time is very small. So another important lesson is to use built-in R functions whenever possible, since they have had substantial development focused on efficiency, correctness, etc.

### 7.5.2 Vectorization

Next consider calculating the sum of the squared entries in each column of a matrix. For example with the matrix  $M$ ,

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

---

<sup>1</sup>Roughly speaking, the first option is slower because each time the vector is increased in size, R must resize the vector and re-allocate memory.

the sums would be  $1^2 + 4^2 = 17$ ,  $2^2 + 5^2 = 29$ , and  $3^2 + 6^2 = 45$ . One possibility is to have an outer loop that traverses the columns and an inner loop that traverses the rows within a column, squaring the entries and adding them together.

```
> test_matrix <- matrix(1:6, byrow = TRUE, nrow = 2)
> test_matrix

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

> ss1 <- function(M) {
+   n <- dim(M)[1]
+   m <- dim(M)[2]
+   out <- rep(0, m)
+   for (j in 1:m) {
+     for (i in 1:n) {
+       out[j] <- out[j] + M[i, j]^2
+     }
+   }
+   return(out)
+ }
> ss1(test_matrix)

[1] 17 29 45
```

Another possibility eliminates the inner loop, using the `sum()` function to compute the sum of the squared entries in the column directly.

```
> ss2 <- function(M) {
+   m <- dim(M)[2]
+   out <- numeric(m)
+   for (j in 1:m) {
+     out[j] <- sum(M[, j]^2)
+   }
+   return(out)
+ }
> ss2(test_matrix)

[1] 17 29 45
```

A third possibility uses the `colSums()` function.

```
> ss3 <- function(M) {
+   out <- colSums(M^2)
+   return(out)
+ }
> ss3(test_matrix)

[1] 17 29 45
```

Here is a speed comparison, using a  $1000 \times 10000$  matrix.

```
> mm <- matrix(1:10000000, byrow=TRUE, nrow=1000)
> system.time(ss1(mm))

  user  system elapsed
20.352   0.008 20.372

> system.time(ss2(mm))

  user  system elapsed
0.168   0.056   0.226

> system.time(ss3(mm))

  user  system elapsed
0.100   0.008   0.107

> rm(mm)
```

## 7.6 More on functions

Understanding functions deeply requires a careful study of R’s scoping rules, as well as a good understanding of environments in R. That’s beyond the scope of this book, but we will briefly discuss some issues that are most salient. For a more in-depth treatment, see “Advanced R” by Hadley Wickham, especially the chapters on functions and environments.

### 7.6.1 Calling functions

When using a function, the functions arguments can be specified in three ways:

- By the full name of the argument.

- By the position of the argument.
- By a partial name of the argument.

```
> tmp_function <- function(first.arg, second.arg, third.arg, fourth.arg) {
+   return(c(first.arg, second.arg, third.arg, fourth.arg))
+ }
> tmp_function(34, 15, third.arg = 11, fou = 99)

[1] 34 15 11 99
```

Positional matching of arguments is convenient, but should be used carefully, and probably limited to the first few, and most commonly used, arguments in a function. And partial matching has pitfalls also. A partially specified argument must unambiguously match exactly one argument—a requirement that’s not met below.

```
> tmp_function <- function(first.arg, fourth.arg) {
+   return(c(first.arg, fourth.arg))
+ }
> tmp_function(1, f = 2)

Error in tmp_function(1, f = 2): argument 2 matches multiple formal arguments
```

## 7.6.2 The ... argument

In defining a function, a special argument denoted by `...` can be used. Sometimes this is called the “ellipsis” argument, sometimes the “three dot” argument, sometimes the “dot dot dot” argument, etc. The R language definition <https://cran.r-project.org/doc/manuals/r-release/R-lang.html> describes the argument in this way:

The special type of argument ‘`...`’ can contain any number of supplied arguments. It is used for a variety of purposes. It allows you to write a function that takes an arbitrary number of arguments. It can be used to absorb some arguments into an intermediate function which can then be extracted by functions called subsequently.

Consider for example the `sum()` function.

```
> sum(1:5)

[1] 15
```

```
> sum(1:5, c(3, 4, 90))
[1] 112

> sum(1, 2, 3, c(3, 4, 90), 1:5)
[1] 118
```

Think about writing such a function. There is no way to predict in advance the number of arguments a user might specify. So the function is defined with `...` as the first argument:

```
> sum
function (..., na.rm = FALSE) .Primitive("sum")
```

This is true of many commonly-used functions in R such as `c()` among others.

Next, consider a function that calls another function in its body. For example, suppose that a collaborator always supplies comma delimited files that have five lines of description, followed by a line containing variable names, followed by the data. You are tired of having to specify `skip = 5`, `header = TRUE`, and `sep = ","` to `read.table()` and want to create a function `my.read()` which uses these as defaults.

```
> my.read <- function(file, header = TRUE, sep = ",", skip = 5,
+   ...) {
+   read.table(file = file, header = header, sep = sep, skip = skip,
+   ...)
+ }
```

The `...` in the definition of `my.read()` allows the user to specify other arguments, for example, `stringsAsFactors = FALSE`. These will be passed on to the `read.table()` function. In fact, that is how `read.csv()` is defined.

```
> read.csv
function (file, header = TRUE, sep = ",", quote = "\"\"", dec = ".",
  fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
  dec = dec, fill = fill, comment.char = comment.char, ...)
<bytecode: 0x000000001bd7ae10>
<environment: namespace:utils>
```

### 7.6.3 Lazy evaluation

Arguments to R functions are not evaluated until they are needed, sometimes called *lazy* evaluation.

```
> f <- function(a, b) {
+   print(a^2)
+   print(a^3)
+   print(a * b)
+ }
> f(a = 3, b = 2)

[1] 9
[1] 27
[1] 6

> f(a = 3)

[1] 9
[1] 27

Error in print(a * b): argument "b" is missing, with no default
```

The first call specified both of the arguments `a` and `b`, and produced the expected output. In the second call the argument `b` was not specified. Since it was not needed until the third `print` statement, R happily executed the first two `print` statements, and only reported an error in the third statement, when `b` was needed to compute `a*b`.

```
> f <- function(a, b = a^3) {
+   return(a * b)
+ }
> f(2)

[1] 16

> f(2, 10)

[1] 20
```

In the first call, since `b` was not specified, it was computed as `a^3`. In the second call, `b` was specified, and the specified value was used.

**Exercise 10**

Learning objectives: translate statistical notation into coded functions; learn about tools for checking validity of function arguments; practice writing functions that return multiple objects.

# Chapter 8

## Simulation

Typically we do not know whether an assumed model for data is correct, and it can be very helpful to learn how procedures for analyzing data are affected if the assumed model is incorrect. Often simulation is a valuable tool in this pursuit.

For two simple but useful examples, consider the  $t$  test and the usual confidence interval for a proportion. The  $t$  test assumes normal population distributions, but how well does it perform if the population distributions are not normal? The confidence interval for a proportion relies on the central limit theorem, and many textbooks state that this asymptotic approximation provides an accurate confidence level if  $n\hat{p}$  and  $n(1 - \hat{p})$  are both 10 or larger. Is this true? Simulation can provide insight into these and many other questions.

Stochastic simulation is a rich and sometimes complex field. This chapter will provide a brief introduction to using R for simulation, but those who need to perform sophisticated simulations will need to learn more!

### 8.1 Pseudo-random numbers

Simulations use numbers generated by a *pseudo-random number generator* (PRNG) rather than by a truly random number generator. Pseudo-random numbers are generated in a way which gives them many important properties of truly random numbers, but they have two important advantages. First, simulations often need the generator to generate many numbers quickly, and PRNGs can be quite efficient. Second, PRNGs generate sequences of numbers which can be reproduced in the future, which helps in conducting simulations. There are applications where truly random number generators are more appropriate, but for simulations PRNGs provide what is needed.

R has a sophisticated PRNG.<sup>1</sup> Although it would take us much too far afield to delve deeply into the PRNG in R, it will help to have an understanding of how one might construct a PRNG. The goal of a PRNG is to generate a sequence of numbers

---

<sup>1</sup>Actually, the R user can choose between several different PRNGs which are built into R. We will just use the default generator.

which “behaves like” a sequence of independent Uniform  $(0,1)$  random variables.

Linear congruential generators are a common starting point. A linear congruential generator requires a multiplier ( $a$ ), an increment ( $c$ ), a modulus ( $m$ ), and a seed  $X_0$ , all integers. The generator is defined by

$$X_i = (aX_{i-1} + c) \mod m.$$

Recall that  $Z \mod m$  gives the remainder when  $Z$  is divided by  $m$ . In R  $Z \mod m$  is implemented as `Z %% m`.

```
> 15%%2
```

```
[1] 1
```

```
> 15%%5
```

```
[1] 0
```

```
> 15%%6
```

```
[1] 3
```

The values  $a, c, X_i$  are all chosen to be in  $\{0, 1, \dots, m - 1\}$  and a pseudo-random sequence from the generator is given by  $U_i = X_i/M$ . Note that the generator will repeat itself after at most  $m$  iterations, i.e., the generator has period at most  $m$ .

The function `lcg()` below implements this idea in R.

```
> lcg <- function(a, c, seed, m, iter) {
+   out <- numeric(iter)
+   out[1] <- seed
+   for (i in 2:(iter)) {
+     out[i] <- (a * out[i - 1] + c)%%m
+   }
+   return(out)
+ }
```

Here are the first few values for several different choices of  $a$  and  $c$ , with the seed always chosen as 0 and the modulus always chosen as 2048.

```
> lcg(a = 65, c = 1, seed = 0, m = 2048, iter = 10)/2048
```

```
[1] 0.0000000000 0.0004882812 0.0322265625 0.0952148438
```

```
[5] 0.1894531250 0.3149414062 0.4716796875 0.6596679688
```

```
[9] 0.8789062500 0.1293945312
```

```
> lcg(a = 1365, c = 1, seed = 0, m = 2048, iter = 10)/2048
[1] 0.0000000000 0.0004882812 0.6669921875 0.4448242188
[5] 0.1855468750 0.2719726562 0.2431640625 0.9194335938
[9] 0.0273437500 0.3247070312

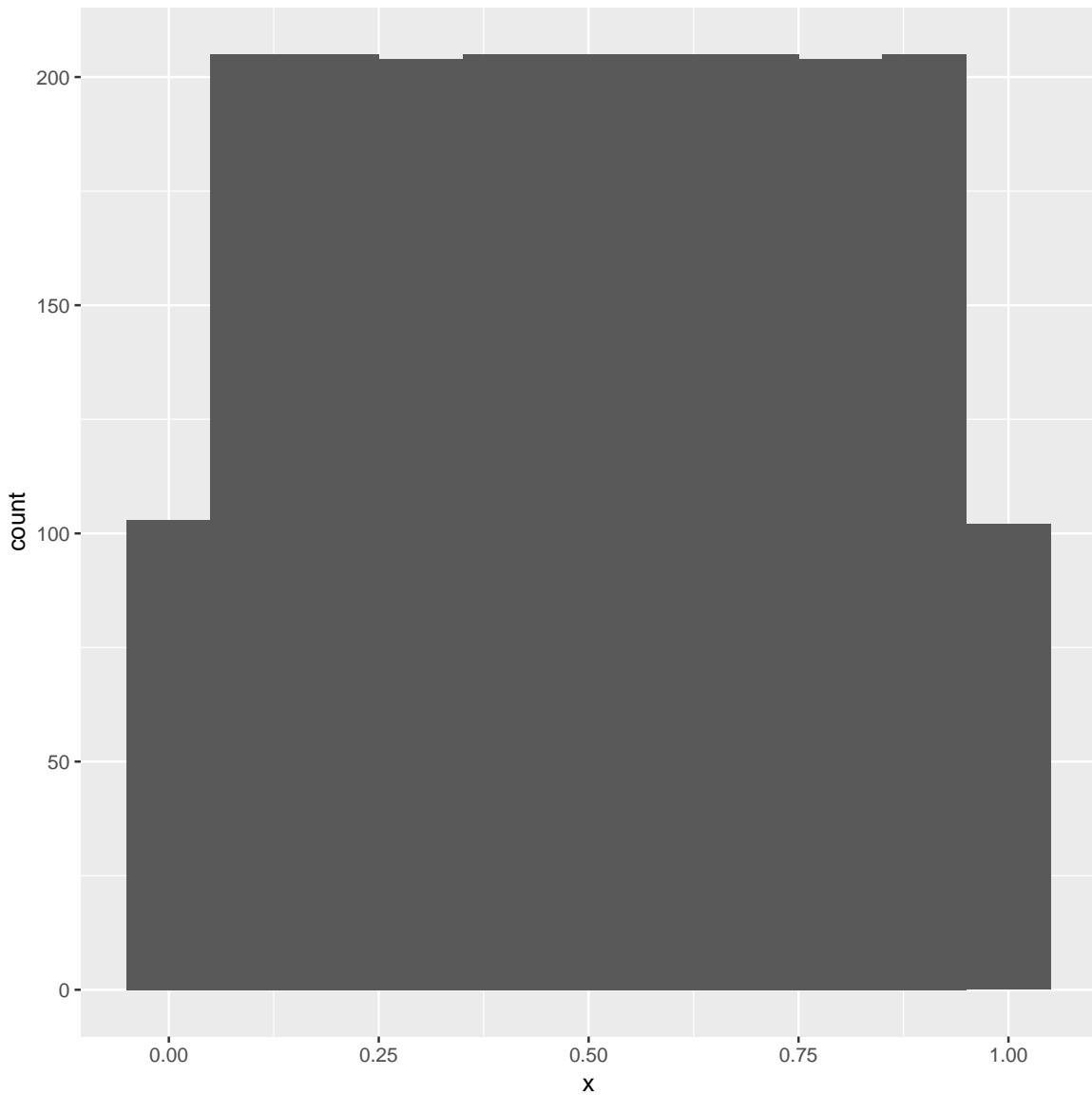
> lcg(a = 1229, c = 1, seed = 0, m = 2048, iter = 10)/2048
[1] 0.0000000000 0.0004882812 0.6005859375 0.1206054688
[5] 0.2246093750 0.0454101562 0.8095703125 0.9624023438
[9] 0.7929687500 0.5590820312

> lcg(a = 157, c = 1, seed = 0, m = 2048, iter = 10)/2048
[1] 0.0000000000 0.0004882812 0.0771484375 0.1127929688
[5] 0.7089843750 0.3110351562 0.8330078125 0.7827148438
[9] 0.8867187500 0.2153320312

> lcg(a = 43, c = 0, seed = 100, m = 2048, iter = 10)/2048
[1] 0.04882812 0.09960938 0.28320312 0.17773438 0.64257812
[6] 0.63085938 0.12695312 0.45898438 0.73632812 0.66210938
```

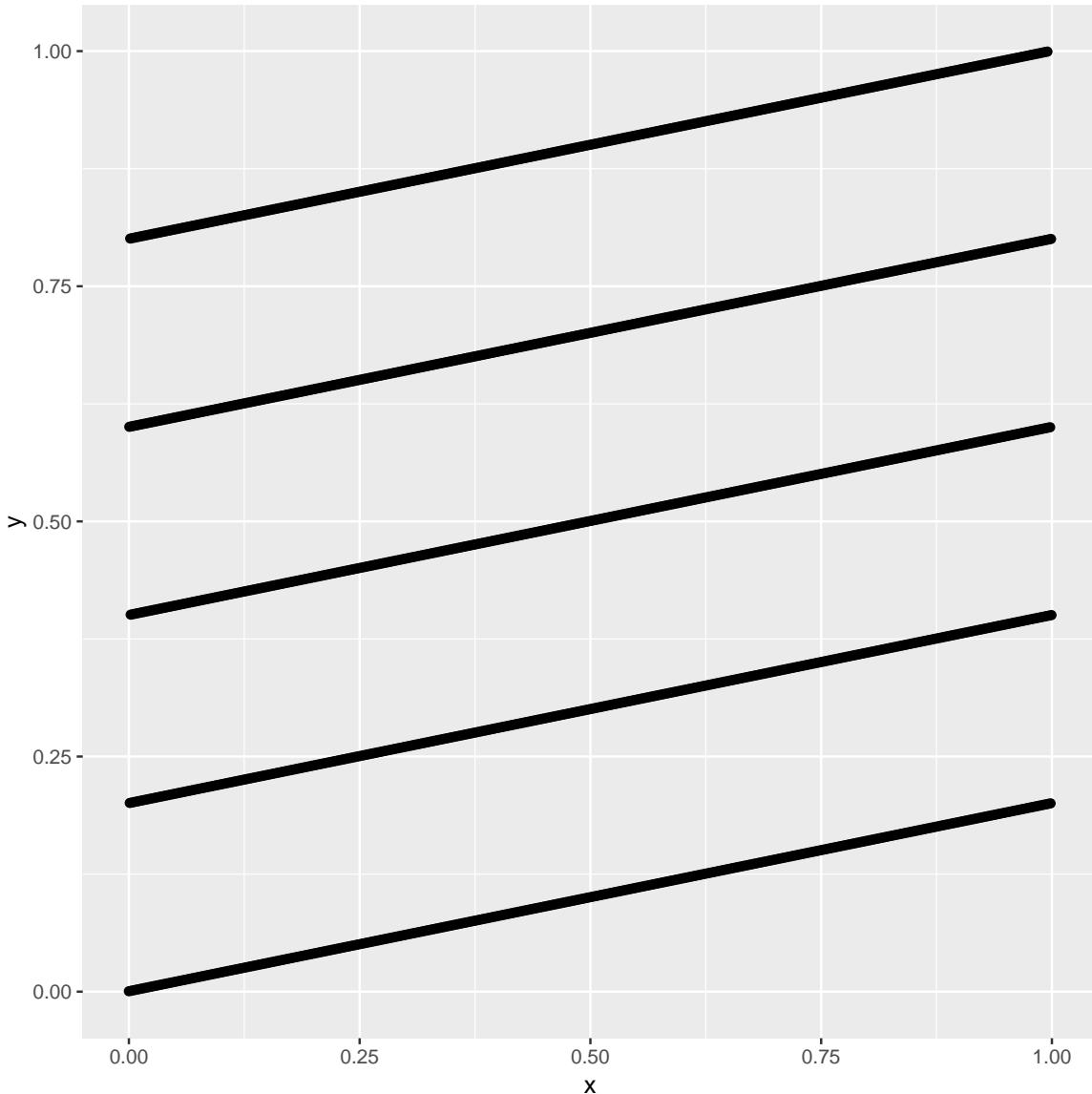
A very basic way to assess the performance of a PRNG is to draw a histogram of the values. The histogram should look like the histogram of uniformly distributed values.

```
> library(ggplot2)
> tmp_random <- data.frame(x = lcg(a = 1229, c = 1, seed = 0, m = 2048,
+     iter = 2048)/2048)
> ggplot(tmp_random, aes(x = x)) + geom_histogram(binwidth = 0.1)
```



This looks good! But of course we want more than just a uniform shape. For example we want the values to be independent. Let's look at a scatter plot of  $U_i$  versus  $U_{i-1}$ .

```
> tmp_random$y <- c(tmp_random$x[2:2048], NA)
> ggplot(tmp_random[1:2047, ], aes(x = x, y = y)) + geom_point()
```



Ouch! That's not what we would like! Ideally the points should be randomly scattered in the unit square.

The point of the preceding examples is that it is NOT easy to create a PRNG, and we are best off using R's built in PRNG, which is implemented in the function `runif()`.

```
> runif(25)

[1] 0.60493329 0.65472393 0.35319727 0.27026015 0.99268406
[6] 0.63349326 0.21320814 0.12937235 0.47811803 0.92407447
[11] 0.59876097 0.97617069 0.73179251 0.35672691 0.43147369
[16] 0.14821156 0.01307758 0.71556607 0.10318424 0.44628435
[21] 0.64010105 0.99183862 0.49559358 0.48434952 0.17344233
```

Being able to provide a seed is important for repeatability of simulations. The `set.seed()` function in R provides this capability. The PRNG in R is much more complex than the linear congruential generators that we created, and the seed has a somewhat more complex relationship with the generator, but what's important is that with the same seed, the generator will produce the same values.

```
> set.seed(123)
> runif(5)

[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673

> set.seed(123)
> runif(5)

[1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673

> runif(5)

[1] 0.0455565 0.5281055 0.8924190 0.5514350 0.4566147
```

## 8.2 Simulating from standard distributions

From a supply  $U_1, U_2, \dots$  of Uniform (0,1) random variables it is possible in principle to generate values from any distribution. There is a wide variety of methods for doing this. For example suppose that we want to generate coin flips, i.e., to generate from a Bernoulli (0.5) distribution. Here is one way to do this.

```
> generate_coin <- function(n) {
+   out <- rep(0, n)
+   out[runif(n) > 0.5] <- 1
+   return(out)
+ }
> generate_coin(10)

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

If we wanted H and T for heads and tails, a similar method would work.

```
> generate_coin <- function(n) {
+   out <- rep("T", n)
+   out[runif(n) > 0.5] <- "H"
```

```
+     return(out)
+ }
> generate_coin(10)

[1] "H" "H" "H" "H" "H" "H" "H" "H" "T" "T"
```

For about twenty common families of distributions, R provides functions that give the density (or mass function), cumulative distribution function, quantile function, and generate values. The functions have a unified naming scheme, illustrated by the functions for the normal distributions.

- **dnorm** is the normal density function
- **pnorm** is the normal cumulative distribution function
- **qnorm** is the normal quantile function
- **rnorm** generates values from a normal distribution.

Rather than designing our own methods for simulating from these distributions, it is better to use the built in functions. Other commonly used families with their **r** functions include the binomial (**rbinom**), geometric (**rgeom**), Poisson (**rpois**), *t* (**rt**), exponential (**rexp**), and hypergeometric (**rhyper**). For the full list use `help("Distributions")`.

```
> # 10 values from the N(100, 2) distribution
> rnorm(10, mean = 100, sd = 2)

[1] 103.57383 100.99570 96.06677 101.40271 99.05442
[6] 97.86435 99.56405 97.94799 98.54222 98.74992

> # 5 values from the Uniform(-3, 4) distribution
> runif(5, min = -3, max = 4)

[1] -2.67918183 0.09540052 2.59247392 -2.14670518
[5] 0.92663589

> # 7 values from the binomial(10, 0.2) distribution
> rbinom(7, size = 10, p = 0.2)

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

In addition, the **sample** function provides a way to sample from a finite population. This function takes up to four arguments: The population **x** to generate values from; the number of values **size** to generate; whether to sample with or without replacement;

and probability weights to use in sampling (by default the weights are  $1/\text{length}(x)$ .)

```
> # 10 coin flips
> sample(c("H", "T"), 10, replace = TRUE)
[1] "H" "H" "T" "H" "T" "T" "T" "H" "T" "T"

> # a random permutation
> sample(1:9, 9, replace = FALSE)
[1] 7 1 4 2 6 3 5 8 9

> # sample from a distribution with P(X=2) = 0.3; P(X=3)=0.5;
> # and P(X=17)=0.2
> sample(c(2, 3, 17), size = 15, replace = TRUE, prob = c(0.3,
+      0.5, 0.2))
[1] 2 3 2 3 3 17 17 17 3 3 2 3 2 3 3
```

## 8.3 Investigating $t$ tests

The one-sample  $t$  test of  $H_0: \mu = \mu_0$  versus  $H_a: \mu \neq \mu_0$  starts with a sample of  $n$  observations  $X_1, \dots, X_n$  and makes use of a  $t$  statistic

$$t = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}.$$

The p-value is twice the area under a  $t$  distribution with  $n - 1$  degrees of freedom to the right of the observed  $t$  statistic. The test is exact under the assumption that the data are i.i.d. (independent and identically distributed) normally distributed.

What if the data are not normally distributed? Is the one-sample  $t$  test still a reasonable procedure to use? We can shed some light on the question by simulating data from various distributions and computing the p-values. Assuming that the null hypothesis is true, the p-values should approximately have a uniform distribution on the interval  $(0, 1)$ , and, in particular, approximately a proportion  $\alpha$  of the p-values should be less than any given  $\alpha \in (0, 1)$ . For example, we should see approximately 0.05 of the p-values less than 0.05.

We will investigate this question for four different distributions: First, a normal distribution with mean 0 and standard deviation 1. Second, an exponential distribution with parameter  $\lambda = 0.1$ , which is rather skewed. Third, a  $t$  distribution with 2 degrees of freedom, which has much heavier tails than a normal distribution. And fourth, a

uniform distribution on  $(-1, 1)$ . And we will simulate for sample sizes  $n = 2$ ,  $n = 5$ ,  $n = 20$ , and  $n = 50$ . In the process we will learn a bit more about functions and graphics.

There are several steps needed to accomplish our goal. First, we need to generate data from the appropriate distribution. Second, we need to compute p-values for the data. Third, we need to draw histograms of the p-values, and possibly report the proportion of p-values less than the standard cutoff of 0.05.

Let's start by writing a function which, given data, computes a p-value from a one-sample  $t$  test.

```
> pval_t <- function(x, mu0 = 0) {
+   tstat <- (mean(x) - mu0)/(sd(x)/sqrt(length(x)))
+   pval <- 2 * (1 - pt(abs(tstat), df = length(x) - 1))
+   return(pval)
+ }
> pval_t(c(1, -2, 4, -3, 3, -5))

[1] 0.8276351
```

Now let's write a function which computes the p-value for a specified number of tests, using the standard normal distribution as the data generating distribution.

```
> simulate_t <- function(sampsize, iter) {
+   pvals <- numeric(iter)
+   for (i in 1:iter) {
+     pvals[i] <- pval_t(rnorm(sampsize, mean = 0, sd = 1))
+   }
+   return(pvals)
+ }
> simulate_t(sampsize = 10, iter = 5)

[1] 0.37964353 0.45150559 0.86860982 0.12632873 0.06298446
```

We could write similar functions for the  $t$  and exponential distributions. But it would be nice to write one function which could be used for any distribution, which could be used for the three distributions of current interest and also could be used in the future for other distributions. For this to work, one of the arguments to the function would itself be a function such as `rnorm` or `rexp`. In addition, the user would have to be able to specify the parameter(s) such as the mean and standard deviation of a normal distribution or the degrees of freedom of a  $t$  distribution. Fortunately R allows functions as arguments, and the `...` argument can be used to allow parameters to be specified. We'll also want to add a `mu0` argument to be passed to the `pval_t` function.

```

> simulate_t <- function(sampszie, mu0 = 0, iter, dist, ...) {
+   pvals <- numeric(iter)
+   for (i in 1:iter) {
+     pvals[i] <- pval_t(dist(sampszie, ...), mu0 = mu0)
+   }
+   return(pvals)
+ }
> simulate_t(sampszie = 2, iter = 10, dist = rnorm, mean = 0, sd = 1)

[1] 0.3563022 0.7208893 0.1070244 0.4698995 0.2199904
[6] 0.7497431 0.9824563 0.4781455 0.1654973 0.6040653

> simulate_t(sampszie = 2, iter = 10, dist = rt, df = 3)

[1] 0.6290604 0.3055575 0.3657688 0.6622394 0.8933037
[6] 0.2524358 0.5419819 0.0304229 0.5073425 0.9768232

> simulate_t(sampszie = 2, mu0 = 10, iter = 10, dist = rexp, rate = 0.1)

[1] 0.91547841 0.81501881 0.24433999 0.16637111 0.18014629
[6] 0.28080653 0.63582295 0.07027922 0.43864056 0.20522431

> simulate_t(sampszie = 2, iter = 10, dist = runif, min = -1, max = 1)

[1] 0.913785696 0.185931808 0.847224312 0.967011562
[5] 0.898950200 0.040169357 0.013736750 0.003375758
[9] 0.964918927 0.901356848

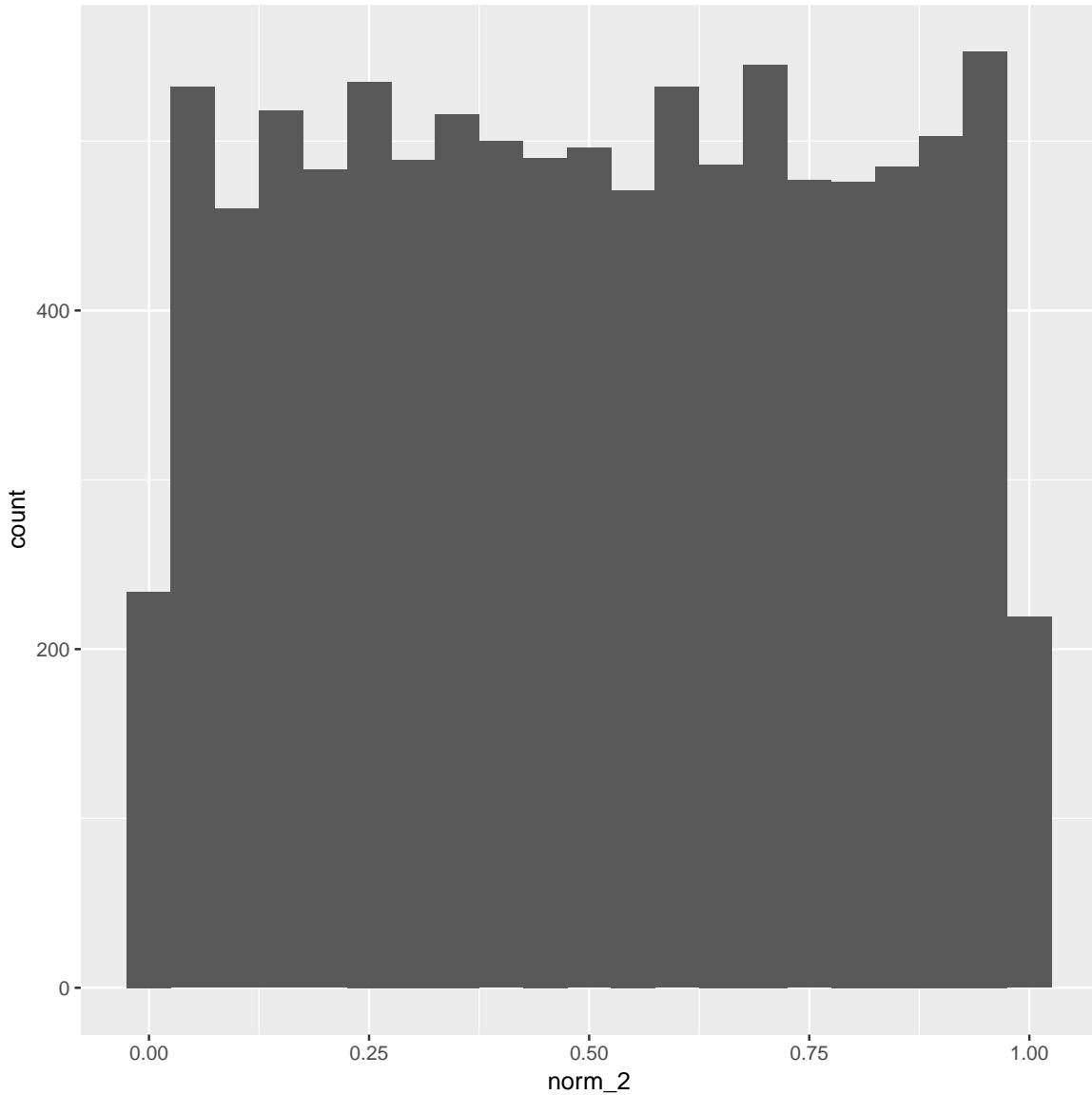
```

Let's use the function for the case of a standard normal data generating distribution with sample size 2.

```

> normal_2 <- data.frame(norm_2 = simulate_t(sampszie = 2, iter = 10000,
+   dist = rnorm, mean = 0, sd = 1))
> ggplot(normal_2, aes(x = norm_2)) + geom_histogram(binwidth = 0.05)

```



```
> with(normal_2, length(norm_2[norm_2 < 0.05])/length(norm_2))
[1] 0.0528
```

The histogram looks roughly normal, and the proportion of  $p$  values which are less than 0.05 is approximately 0.05, as expected.

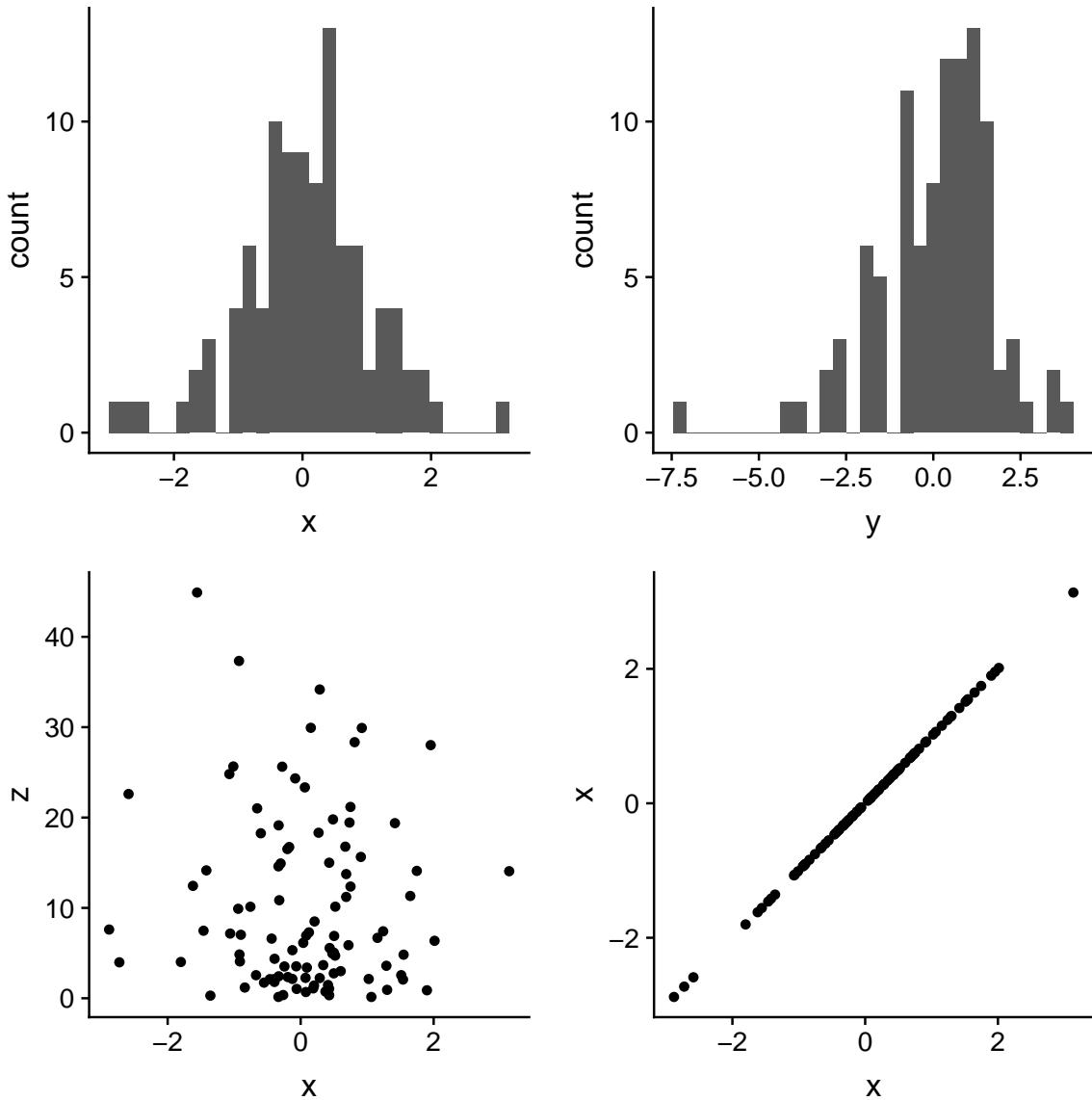
Before conducting the simulation, it would be nice to find a way to show more than one histogram on a grid. There are several ways to accomplish this. One that is both simple and powerful is included in the `cowplot` package, and we will use it. Here is a simple example of its use.

```
> library(cowplot)

Attaching package: 'cowplot'
The following object is masked from 'package:ggplot2':
  ggsave

> dd <- data.frame(x = rnorm(100), y = rt(100, 2), z = rexp(100,
+      0.1), w = runif(100, 0, 1))
> p1 <- ggplot(dd, aes(x = x)) + geom_histogram()
> p2 <- ggplot(dd, aes(x = y)) + geom_histogram()
> p3 <- ggplot(dd, aes(x = x, y = z)) + geom_point()
> p4 <- ggplot(dd, aes(x = x, y = x)) + geom_point()
> plot_grid(p1, p2, p3, p4, nrow = 2, ncol = 2)

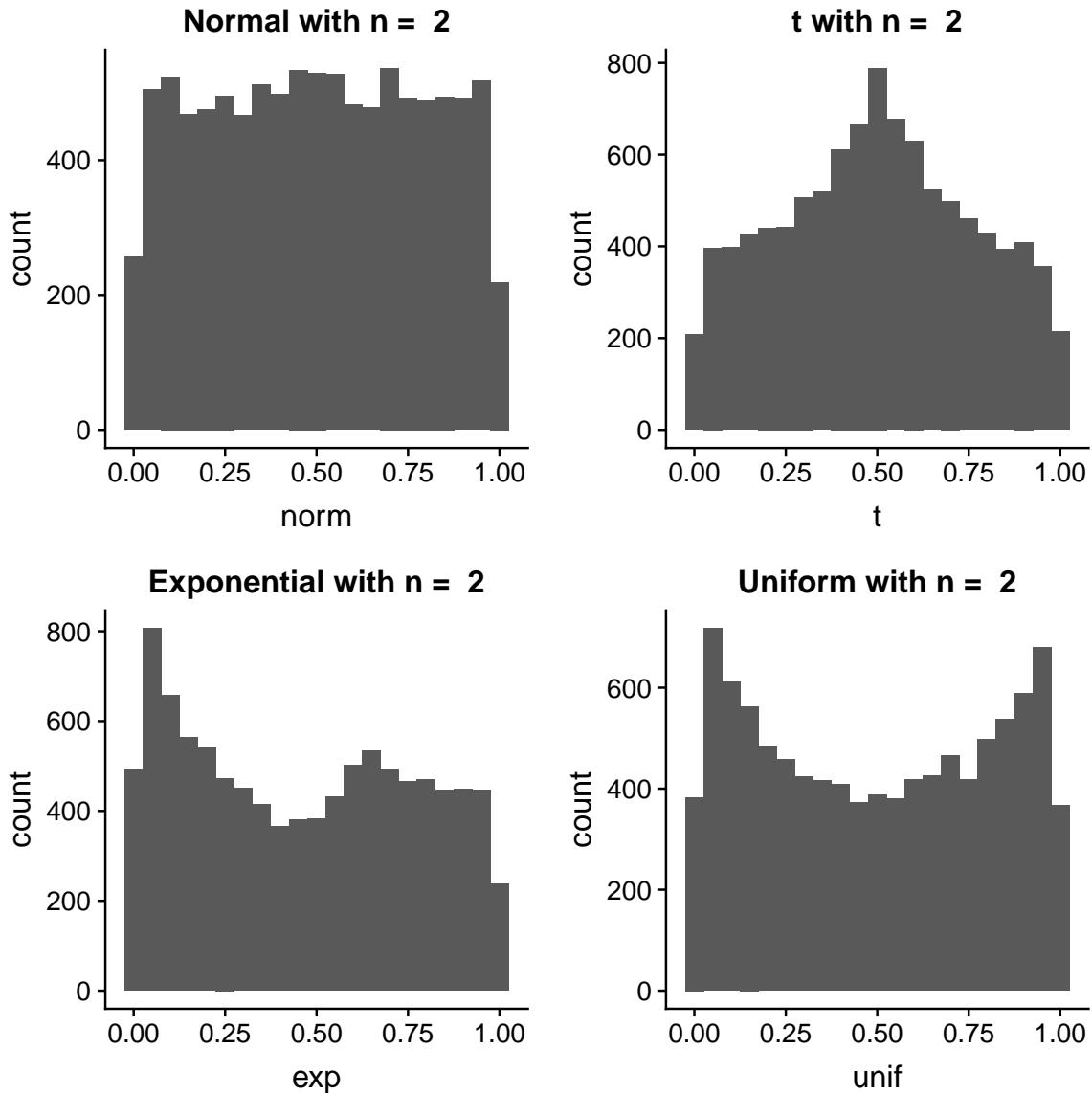
'stat_bin()' using 'bins = 30'. Pick better value with
'binwidth'.
'stat_bin()' using 'bins = 30'. Pick better value with
'binwidth'.
```



Here is the simulation for  $n = 2$ . Note that we set the seed before beginning the simulation.

```
> library(cowplot)
> set.seed(12345)
> iterations <- 10000
> n <- 2
> t_test_simulation <- data.frame(norm = numeric(iterations), t = numeric(iterations),
+ exp = numeric(iterations), unif = numeric(iterations))
> t_test_simulation$norm <- simulate_t(sampszie = n, iter = iterations,
+ dist = rnorm, mean = 0, sd = 1)
> t_test_simulation$t <- simulate_t(sampszie = n, iter = iterations,
+ dist = rt, df = 2)
```

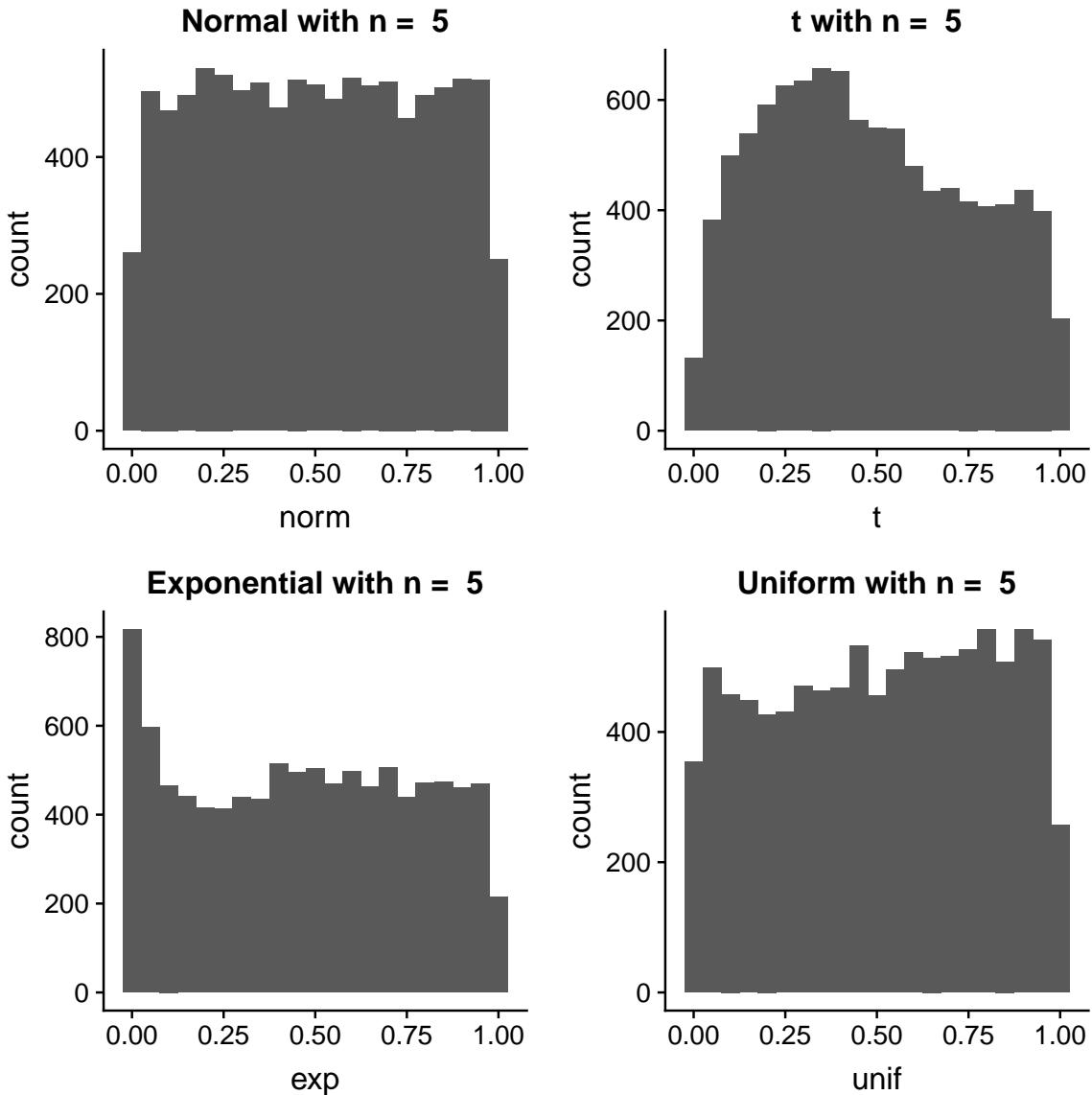
```
> t_test_simulation$exp <- simulate_t(sampszie = n, mu0 = 10, iter = iterations,
+     dist = rexp, rate = 0.1)
> t_test_simulation$unif <- simulate_t(sampszie = n, iter = iterations,
+     dist = runif, min = -1, max = 1)
> p1 <- ggplot(t_test_simulation, aes(x = norm)) + geom_histogram(binwidth = 0.05) +
+     ggttitle(paste("Normal with n = ", n))
> p2 <- ggplot(t_test_simulation, aes(x = t)) + geom_histogram(binwidth = 0.05) +
+     ggttitle(paste("t with n = ", n))
> p3 <- ggplot(t_test_simulation, aes(x = exp)) + geom_histogram(binwidth = 0.05) +
+     ggttitle(paste("Exponential with n = ", n))
> p4 <- ggplot(t_test_simulation, aes(x = unif)) + geom_histogram(binwidth = 0.05) +
+     ggttitle(paste("Uniform with n = ", n))
> plot_grid(p1, p2, p3, p4, nrow = 2, ncol = 2)
```



```
> with(t_test_simulation, length(norm[norm < 0.05])/length(norm))
[1] 0.0496
> with(t_test_simulation, length(t[t < 0.05])/length(t))
[1] 0.0403
> with(t_test_simulation, length(exp[exp < 0.05])/length(exp))
[1] 0.0926
> with(t_test_simulation, length(unif[unif < 0.05])/length(unif))
[1] 0.0764
```

Next, the simulation with  $n = 5$ .

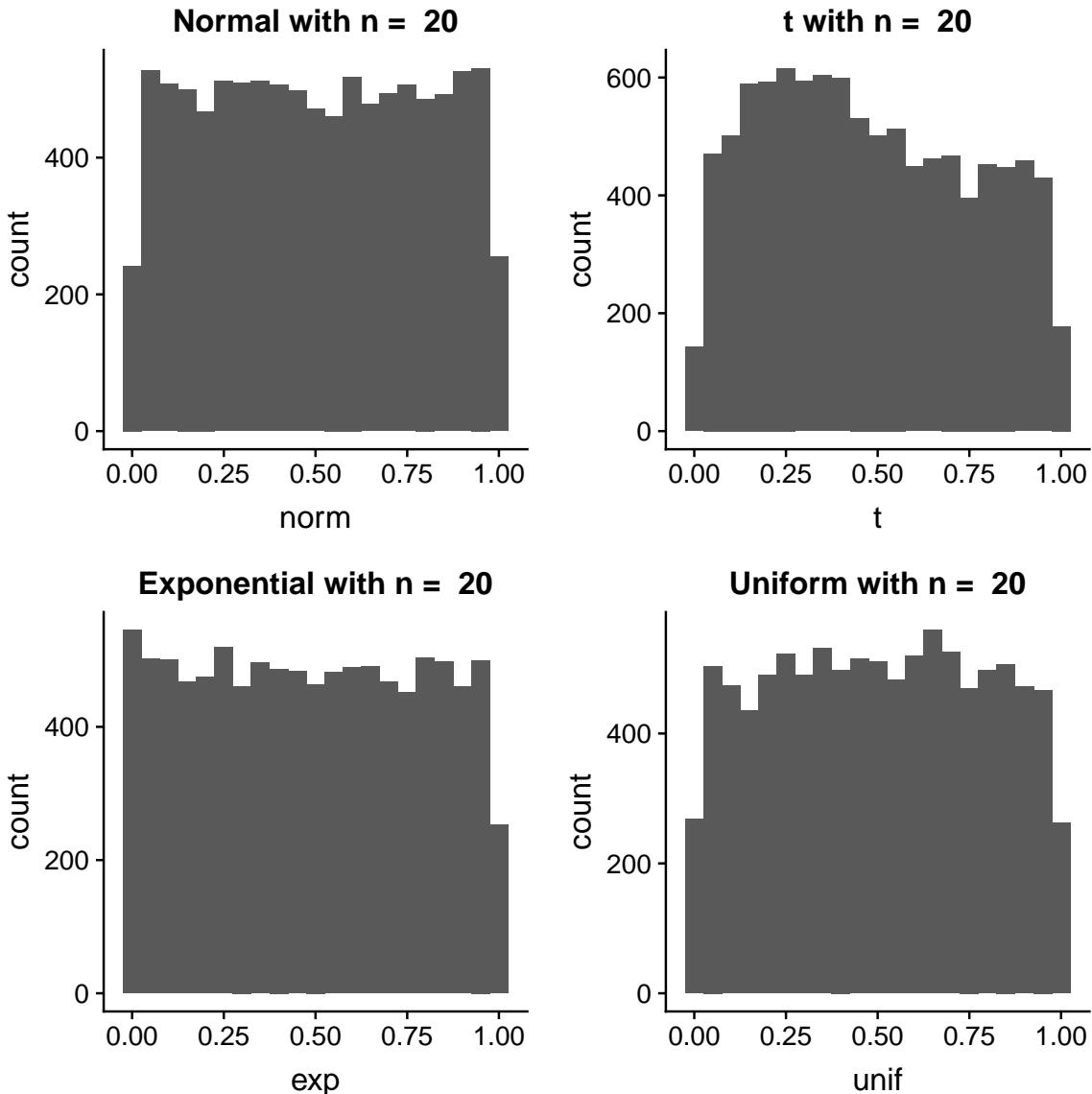
```
> library(cowplot)
> n <- 5
> t_test_simulation$norm <- simulate_t(sampsize = n, iter = iterations,
+   dist = rnorm, mean = 0, sd = 1)
> t_test_simulation$t <- simulate_t(sampsize = n, iter = iterations,
+   dist = rt, df = 2)
> t_test_simulation$exp <- simulate_t(sampsize = n, mu0 = 10, iter = iterations,
+   dist = rexp, rate = 0.1)
> t_test_simulation$unif <- simulate_t(sampsize = n, iter = iterations,
+   dist = runif, min = -1, max = 1)
> p1 <- ggplot(t_test_simulation, aes(x = norm)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Normal with n = ", n))
> p2 <- ggplot(t_test_simulation, aes(x = t)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("t with n = ", n))
> p3 <- ggplot(t_test_simulation, aes(x = exp)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Exponential with n = ", n))
> p4 <- ggplot(t_test_simulation, aes(x = unif)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Uniform with n = ", n))
> plot_grid(p1, p2, p3, p4, nrow = 2, ncol = 2)
```



```
> with(t_test_simulation, length(norm[norm < 0.05])/length(norm))
[1] 0.049
> with(t_test_simulation, length(t[t < 0.05])/length(t))
[1] 0.0305
> with(t_test_simulation, length(exp[exp < 0.05])/length(exp))
[1] 0.1159
> with(t_test_simulation, length(unif[unif < 0.05])/length(unif))
[1] 0.0605
```

Next, the simulation with  $n = 20$ .

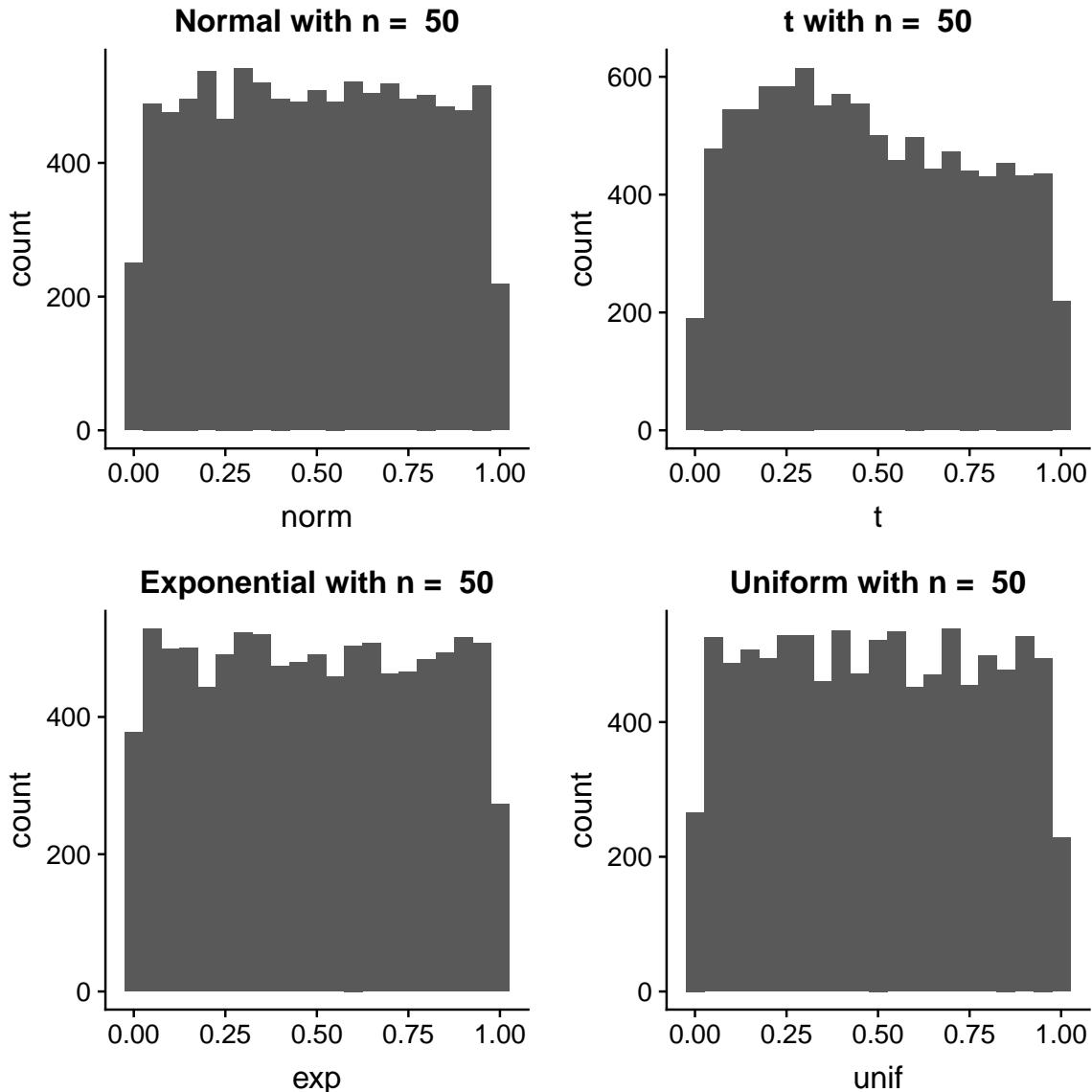
```
> library(cowplot)
> n <- 20
> t_test_simulation$norm <- simulate_t(sampsize = n, iter = iterations,
+   dist = rnorm, mean = 0, sd = 1)
> t_test_simulation$t <- simulate_t(sampsize = n, iter = iterations,
+   dist = rt, df = 2)
> t_test_simulation$exp <- simulate_t(sampsize = n, mu0 = 10, iter = iterations,
+   dist = rexp, rate = 0.1)
> t_test_simulation$unif <- simulate_t(sampsize = n, iter = iterations,
+   dist = runif, min = -1, max = 1)
> p1 <- ggplot(t_test_simulation, aes(x = norm)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Normal with n = ", n))
> p2 <- ggplot(t_test_simulation, aes(x = t)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("t with n = ", n))
> p3 <- ggplot(t_test_simulation, aes(x = exp)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Exponential with n = ", n))
> p4 <- ggplot(t_test_simulation, aes(x = unif)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Uniform with n = ", n))
> plot_grid(p1, p2, p3, p4, nrow = 2, ncol = 2)
```



```
> with(t_test_simulation, length(norm[norm < 0.05])/length(norm))
[1] 0.051
> with(t_test_simulation, length(t[t < 0.05])/length(t))
[1] 0.0371
> with(t_test_simulation, length(exp[exp < 0.05])/length(exp))
[1] 0.08
> with(t_test_simulation, length(unif[unif < 0.05])/length(unif))
[1] 0.0541
```

Next, the simulation with  $n = 50$ .

```
> library(cowplot)
> n <- 50
> t_test_simulation$norm <- simulate_t(sampsize = n, iter = iterations,
+   dist = rnorm, mean = 0, sd = 1)
> t_test_simulation$t <- simulate_t(sampsize = n, iter = iterations,
+   dist = rt, df = 2)
> t_test_simulation$exp <- simulate_t(sampsize = n, mu0 = 10, iter = iterations,
+   dist = rexp, rate = 0.1)
> t_test_simulation$unif <- simulate_t(sampsize = n, iter = iterations,
+   dist = runif, min = -1, max = 1)
> p1 <- ggplot(t_test_simulation, aes(x = norm)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Normal with n = ", n))
> p2 <- ggplot(t_test_simulation, aes(x = t)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("t with n = ", n))
> p3 <- ggplot(t_test_simulation, aes(x = exp)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Exponential with n = ", n))
> p4 <- ggplot(t_test_simulation, aes(x = unif)) + geom_histogram(binwidth = 0.05) +
+   ggtitle(paste("Uniform with n = ", n))
> plot_grid(p1, p2, p3, p4, nrow = 2, ncol = 2)
```



```
> with(t_test_simulation, length(norm[norm < 0.05])/length(norm))
[1] 0.0489
> with(t_test_simulation, length(t[t < 0.05])/length(t))
[1] 0.0412
> with(t_test_simulation, length(exp[exp < 0.05])/length(exp))
[1] 0.0647
> with(t_test_simulation, length(unif[unif < 0.05])/length(unif))
[1] 0.0527
```

From the simulations we see that for small sample sizes, the  $t$  test does not perform well for all three non-normal distributions. As the sample size increases, the performance improves (due to the central limit theorem) but is still somewhat poor even when  $n = 50$  for the highly skewed exponential (0.1) distribution.

## 8.4 Exercise

### Exercise 11

Learning objectives: explore the logistic regression classification method; apply the kNN classification method; create confusion matrices to compare classification methods; plot classified data.

# Chapter 9

## Classification

Classification problems are quite common. For example a spam filter is asked to classify incoming messages into spam or non-spam, based on factors such as the sender’s address, the subject of the message, the contents of the message, and so on. As another example, a doctor diagnosing a patient into one of four possible diagnoses, based on symptoms, blood tests, and medical history, is performing a classification. Or a bank may want to determine (prior to giving a loan) whether an applicant for a loan will default on the loan, again based on several variables such as income, financial history, etc.

Classification methods are both extremely useful and an active area of research in statistics. In this chapter we will learn about two common, and somewhat different, classification methods, logistic regression and  $k$  nearest neighbors. A very good introduction to classification and many other “Statistical Learning” methods is [James et al. \[2014\]](#). The abbreviated treatment in this chapter draws from [James et al. \[2014\]](#).

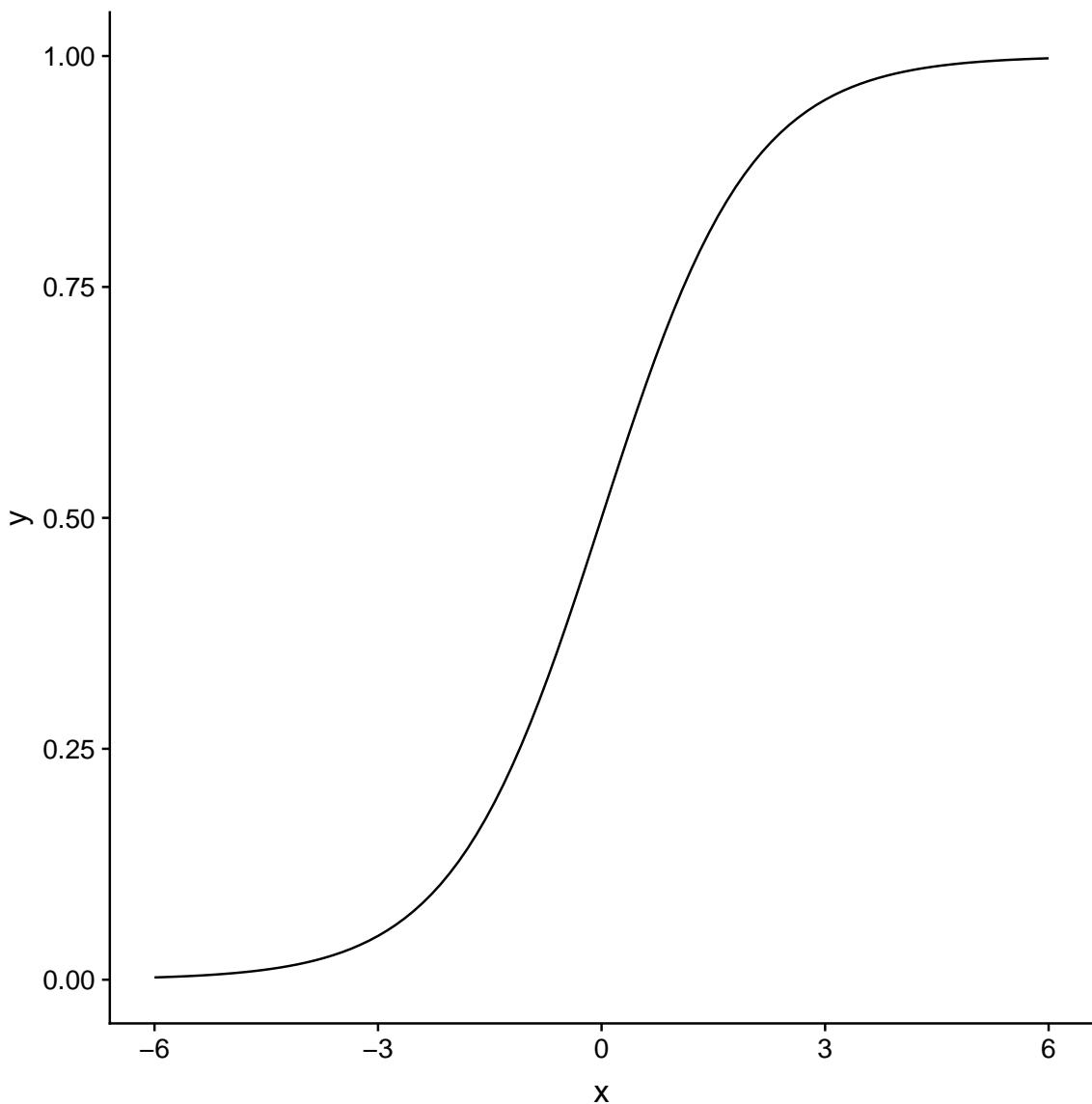
### 9.1 Logistic regression

Logistic regression is widely used to relate a categorical response variable to one or more (continuous or categorical) predictors. Initially we will consider the simplest case where the response  $Y$  has only two possible values (we’ll assume the values are 0 and 1) and where there is only one continuous predictor  $X$ . We would like to predict the value of  $Y$  based on the value of the predictor  $X$ . Let  $p(X) = P(Y = 1|X)$ . We will model  $p(X)$  with the *logistic function*, which takes values between 0 and 1, which is of course appropriate for modeling a probability:

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}.$$

A graph of this function when  $\beta_0 = 0$  and  $\beta_1 = 1$  shows the characteristic shape.

```
> library(ggplot2)
> logistic <- function(x) {
+   exp(x)/(1 + exp(x))
+ }
> ggplot(data.frame(x = c(-6, 6)), aes(x)) + stat_function(fun = logistic)
```



To make this more concrete, we will consider a dataset on a population of women who were over 21 years of age, lived near Phoenix, Arizona, and were of Pima Indian heritage. The data included diabetes status (yes or no) and seven possible predictor variables such as age, number of pregnancies, body mass index, etc. The original data are from the National Institute of Diabetes and Digestive and Kidney Diseases. These data were cleaned and are available in the MASS package in R. In the package the data frame `Pima.tr` contains 200 randomly selected cases from the full data set, which we

will use to find a classifier. The data frame `Pima.te` contains the remaining 332 cases, which we will use to test the classifier on new data.

```
> library(MASS)
> head(Pima.tr)

  npreg  glu  bp  skin  bmi   ped  age type
1     5  86  68    28 30.2 0.364  24   No
2     7 195  70    33 25.1 0.163  55  Yes
3     5  77  82    41 35.8 0.156  35   No
4     0 165  76    43 47.9 0.259  26   No
5     0 107  60    25 26.4 0.133  23   No
6     5  97  76    27 35.6 0.378  52  Yes
```

It will be more convenient to code the presence or absence of diabetes by 1 and 0, so we first create another column in the data frame with this coding:

```
> Pima.tr$diabetes <- rep(0, dim(Pima.tr)[1])
> Pima.tr$diabetes[Pima.tr$type == "Yes"] <- 1
> head(Pima.tr)

  npreg  glu  bp  skin  bmi   ped  age type diabetes
1     5  86  68    28 30.2 0.364  24   No      0
2     7 195  70    33 25.1 0.163  55  Yes      1
3     5  77  82    41 35.8 0.156  35   No      0
4     0 165  76    43 47.9 0.259  26   No      0
5     0 107  60    25 26.4 0.133  23   No      0
6     5  97  76    27 35.6 0.378  52  Yes      1
```

We will begin with `glu` as a predictor, and of course `type` as the response. So we want to find the values  $\beta_0$  and  $\beta_1$  which provide the best fit for the model

$$P(\text{yes}|\text{glu}) = \frac{e^{\beta_0 + \beta_1(\text{glu})}}{1 + e^{\beta_0 + \beta_1(\text{glu})}}$$

Usually maximum likelihood methods are used to fit the model.

In R we will use the function `glm` to fit logistic regression models. The `glm` function fits a wide variety of models. To specify the logistic regression model we specify `family = binomial` as an argument to `glm`. We also must specify the predictor and response variables via the model formula, which in our case will be `diabetes ~ glu` to indicate that `diabetes` (i.e., `type` recoded) is the response and `glu` is the predictor; also we specify the data frame, which in our case is `Pima.tr`.

```

> diabetes.lri <- glm(diabetes ~ glu, data = Pima.tr, family = binomial)
> diabetes.lri

Call: glm(formula = diabetes ~ glu, family = binomial, data = Pima.tr)

Coefficients:
(Intercept)      glu
-5.50364      0.03778

Degrees of Freedom: 199 Total (i.e. Null); 198 Residual
Null Deviance: 256.4
Residual Deviance: 207.4 AIC: 211.4

> summary(diabetes.lri)

Call:
glm(formula = diabetes ~ glu, family = binomial, data = Pima.tr)

Deviance Residuals:
    Min      1Q  Median      3Q     Max
-1.9714 -0.7795 -0.5292  0.8491  2.2633

Coefficients:
            Estimate Std. Error z value     Pr(>|z|)
(Intercept) -5.503636   0.836077 -6.583 0.000000000462
glu         0.037784   0.006278  6.019 0.0000000017561

(Intercept) ***
glu          ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 256.41 on 199 degrees of freedom
Residual deviance: 207.37 on 198 degrees of freedom
AIC: 211.37

Number of Fisher Scoring iterations: 4

> beta0.lri.1 <- coef(diabetes.lri)[1]

```

```
> beta1.lr.1 <- coef(diabetes.lr1)[2]
> beta0.lr.1

(Intercept)
-5.503636

> beta1.lr.1

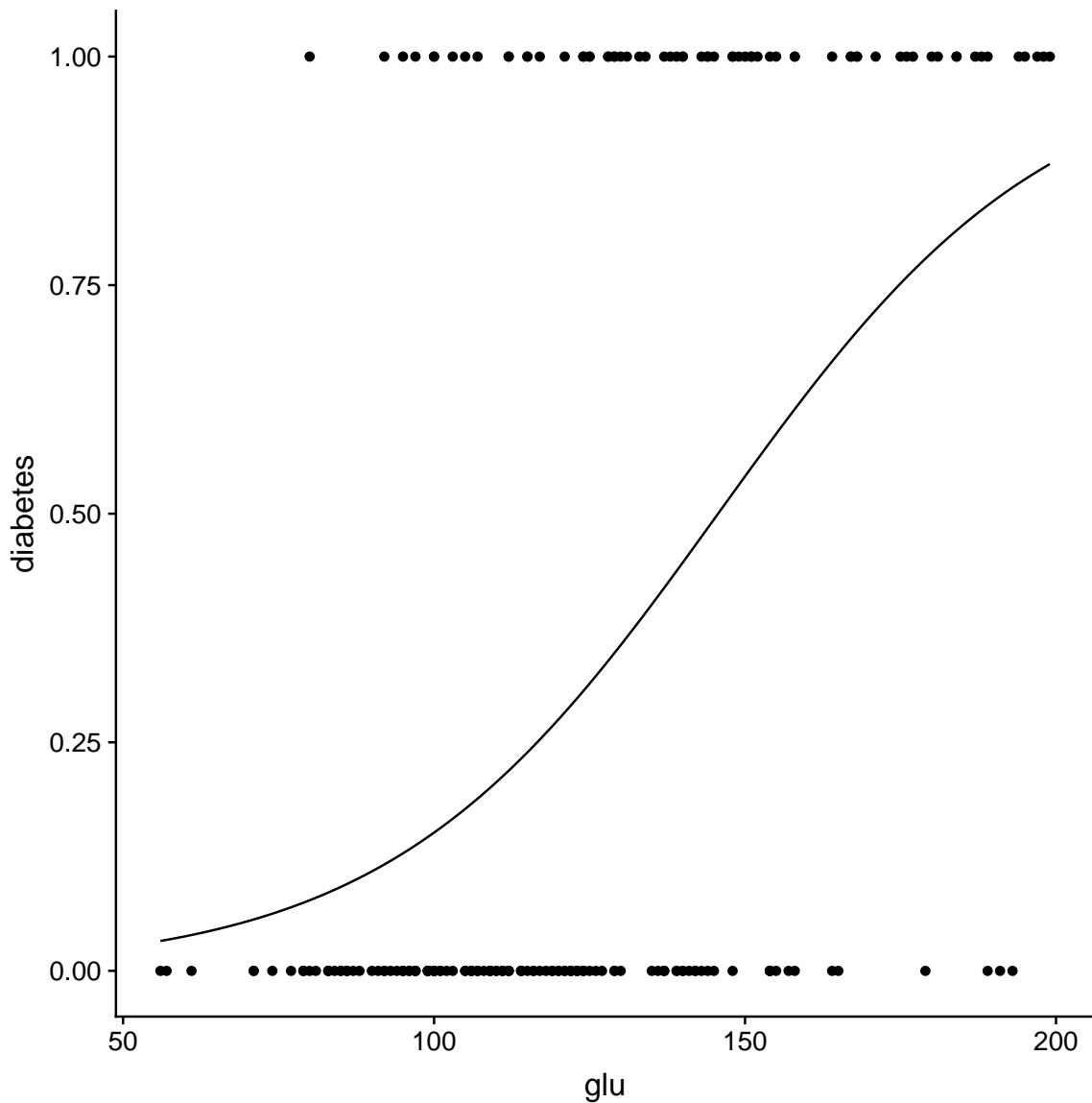
      glu
0.03778372
```

The coefficients  $\beta_0$  and  $\beta_1$  are approximately  $\beta_0 = -5.504$  and  $\beta_1 = 0.038$ . So for example we can estimate the probability that a woman in this population whose glucose level is 150 would be diabetic as

```
> exp(-5.504 + 0.038 * 150)/(1 + exp(-5.504 + 0.038 * 150))
[1] 0.5488437
```

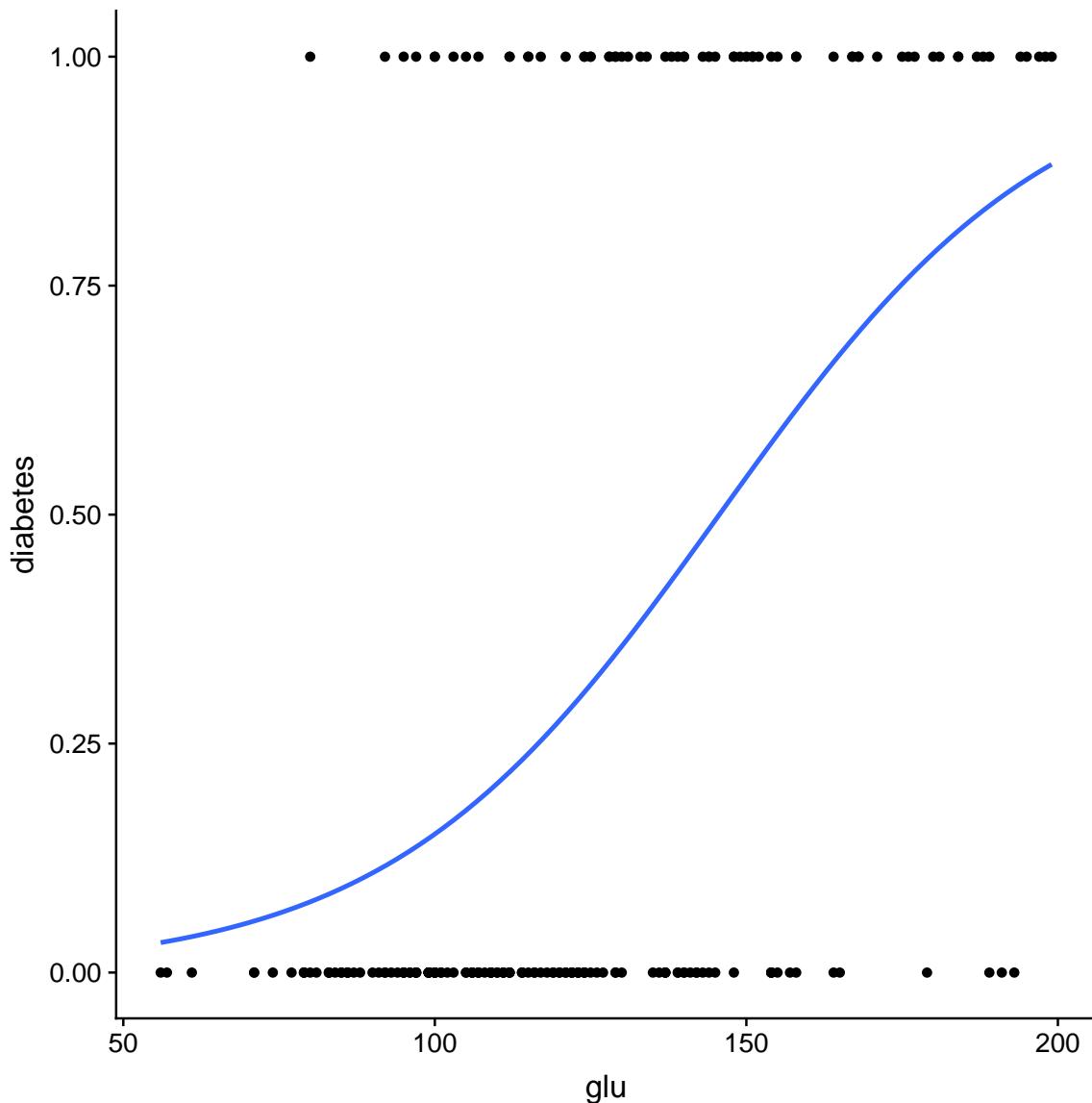
We can plot the fitted probabilities along with the data “by hand.”

```
> diabetes.logistic.1 <- function(x) {
+   exp(beta0.lr.1 + beta1.lr.1 * x)/(1 + exp(beta0.lr.1 + beta1.lr.1 *
+     x))
+ }
> ggplot(Pima.tr, aes(x = glu, y = diabetes)) + stat_function(fun = diabetes.logistic.1)
+   geom_point()
```



The `ggplot2` package also provides a way to do this more directly, using `stat_smooth`.

```
> ggplot(Pima.tr, aes(x = glu, y = diabetes)) + geom_point() +
+   stat_smooth(method = "glm", method.args = list(family = "binomial"),
+              se = FALSE)
```



From these graphics we can see that although glucose level and diabetes are related, there are many women with high glucose levels who are not diabetic, and many with low glucose levels who are diabetic, so likely adding other predictors to the model will help.

Next let's see how the model does in predicting diabetes status in the data we did not use for fitting the model. We will predict diabetes for anyone whose glucose level leads to a model-based probability greater than 1/2. First we use the `predict` function to compute the probabilities, and then use these to make predictions.

```
> head(Pima.te)

  npreg glu bp skin  bmi   ped age type
1     6 148 72    35 33.6 0.627 50  Yes
```

```

2      1 85 66   29 26.6 0.351  31   No
3      1 89 66   23 28.1 0.167  21   No
4      3 78 50   32 31.0 0.248  26   Yes
5      2 197 70  45 30.5 0.158  53   Yes
6      5 166 72  19 25.8 0.587  51   Yes

> diabetes.probs.1 <- predict(diabetes.lri, Pima.te, type = "response")
> head(diabetes.probs.1)

      1          2          3          4          5
0.52207428 0.09178605 0.10518608 0.07199064 0.87432542
      6
0.68318799

```

The `predict` function (with `type = "response"` specified) provides  $p(x) = P(Y = 1|X = x)$  for all the  $x$  values in a data frame. In this case we specified the data frame `Pima.te` since we want to know how the model does in predicting diabetes in a new population, i.e., in a population that wasn't used to “train” the model.

```

> diabetes.predict.1 <- rep("No", dim(Pima.te)[1])
> diabetes.predict.1[diabetes.probs.1 > 0.5] <- "Yes"
> head(diabetes.predict.1)

[1] "Yes" "No"  "No"  "No"  "Yes" "Yes"

> table(diabetes.predict.1, Pima.te$type)

diabetes.predict.1  No Yes
                  No 206 58
                  Yes 17 51

> length(diabetes.predict.1[diabetes.predict.1 == Pima.te$type])/dim(Pima.te)[1]

[1] 0.7740964

```

The table (sometimes called a *confusion matrix*) has the predictions of the model in the rows, so for example we see that the model predicts that  $206 + 58 = 264$  of the women will not be diabetic, and that  $17 + 51 = 68$  of the women will be diabetic. More interesting of course are the cells themselves. For example, of the  $206 + 17 = 223$  women who are not diabetic in `Pima.te`, the model correctly classifies 206, and misclassifies 17. A classifier that predicted perfectly for the test data would have zeros off the diagonal.

### 9.1.1 Adding predictors

If we have  $p$  predictors  $X = (X_1, \dots, X_p)$ , the logistic model becomes

$$p(X) = p(X) = \frac{e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \dots + \beta_p X_p}}.$$

Although there is a lot more notation to keep track of, the basic ideas are the same as they were for the one predictor model. We will next see how adding `bmi`, the body mass index, affects predictions of diabetes.

```
> diabetes.lr2 <- glm(diabetes ~ glu + bmi, data = Pima.tr, family = binomial)
> diabetes.lr2

Call: glm(formula = diabetes ~ glu + bmi, family = binomial, data = Pima.tr)

Coefficients:
(Intercept)          glu          bmi
-8.21611        0.03572       0.09002

Degrees of Freedom: 199 Total (i.e. Null); 197 Residual
Null Deviance: 256.4
Residual Deviance: 198.5 AIC: 204.5

> summary(diabetes.lr2)

Call:
glm(formula = diabetes ~ glu + bmi, family = binomial, data = Pima.tr)

Deviance Residuals:
    Min      1Q      Median      3Q      Max 
-2.0577 -0.7566 -0.4313  0.8011  2.2489 

Coefficients:
            Estimate Std. Error z value     Pr(>|z|)    
(Intercept) -8.216106   1.346965 -6.100 0.00000000106 ***
glu         0.035716   0.006311  5.659 0.00000001520 ***
bmi         0.090016   0.031268  2.879     0.00399 **  
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 ' ' 1

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

```
Null deviance: 256.41 on 199 degrees of freedom
Residual deviance: 198.47 on 197 degrees of freedom
AIC: 204.47
```

```
Number of Fisher Scoring iterations: 4
```

Now we look at predictions from this model.

```
> diabetes.probs.2 <- predict(diabetes.lr2, Pima.te, type = "response")
> head(diabetes.probs.2)

      1          2          3          4          5
0.52358599 0.05809584 0.07530477 0.06662362 0.82713369
      6
0.50879270

> diabetes.predict.2 <- rep("No", dim(Pima.te)[1])
> diabetes.predict.2[diabetes.probs.2 > 0.5] <- "Yes"
> head(diabetes.predict.2)

[1] "Yes" "No"  "No"  "No"  "Yes" "Yes"

> table(diabetes.predict.2, Pima.te$type)

diabetes.predict.2  No Yes
      No    204   54
      Yes    19   55

> length(diabetes.predict.2[diabetes.predict.2 == Pima.te$type])/dim(Pima.te)[1]

[1] 0.7801205
```

Adding `bmi` as a predictor did not improve the predictions by very much!

Let  $x_1$  and  $x_2$  represent glucose and bmi levels. We classify a subject as “diabetic” if the fitted  $p(X)$  is greater than 0.5, i.e., if the fitted probability of diabetes is greater than the fitted probability of not diabetes. The boundary for our decision is where these two fitted probabilities are equal, i.e., where

$$\frac{P(Y = 1|(x_1, x_2))}{P(Y = 0|(x_1, x_2))} = \frac{P(Y = 1|(x_1, x_2))}{1 - P(Y = 1|(x_1, x_2))} = 1.$$

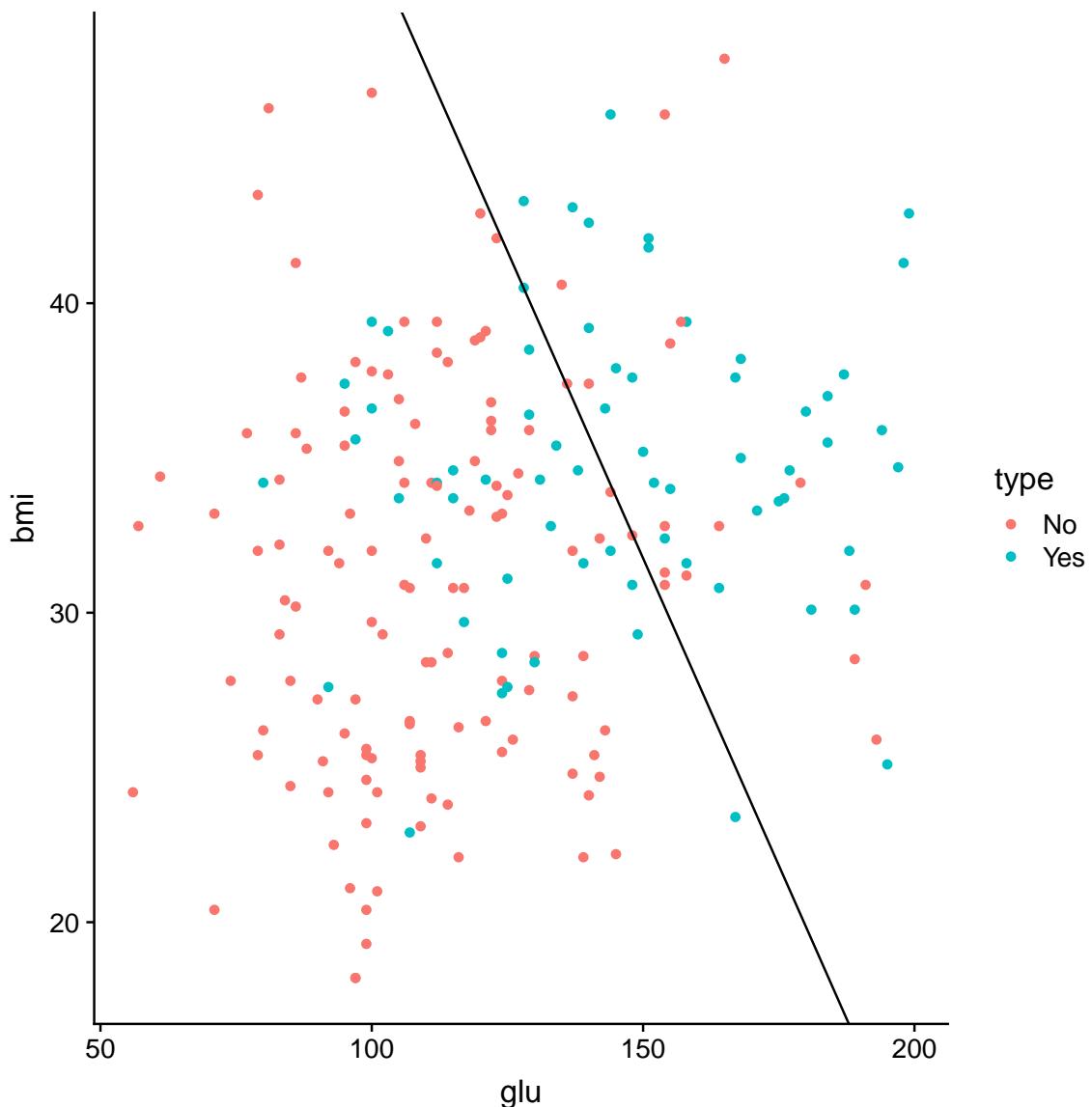
Writing these out in terms of the logistic regression model, taking logarithms, and

performing some algebra leads to the following (linear!) decision boundary:

$$x_2 = -\frac{\beta_0}{\beta_2} - \frac{\beta_1}{\beta_2}x_1.$$

The diabetes training data along with the decision boundary are plotted below.

```
> lr.int <- -coef(diabetes.lr2)[1]/coef(diabetes.lr2)[3]
> lr.slope <- -coef(diabetes.lr2)[2]/coef(diabetes.lr2)[3]
> ggplot(Pima.tr, aes(x = glu, y = bmi)) + geom_point(aes(color = type)) +
+   geom_abline(intercept = lr.int, slope = lr.slope)
```



### 9.1.2 More than two classes

Logistic regression methods also are applicable to classification contexts where there are more than two classes. Consider for example Fisher's iris data.

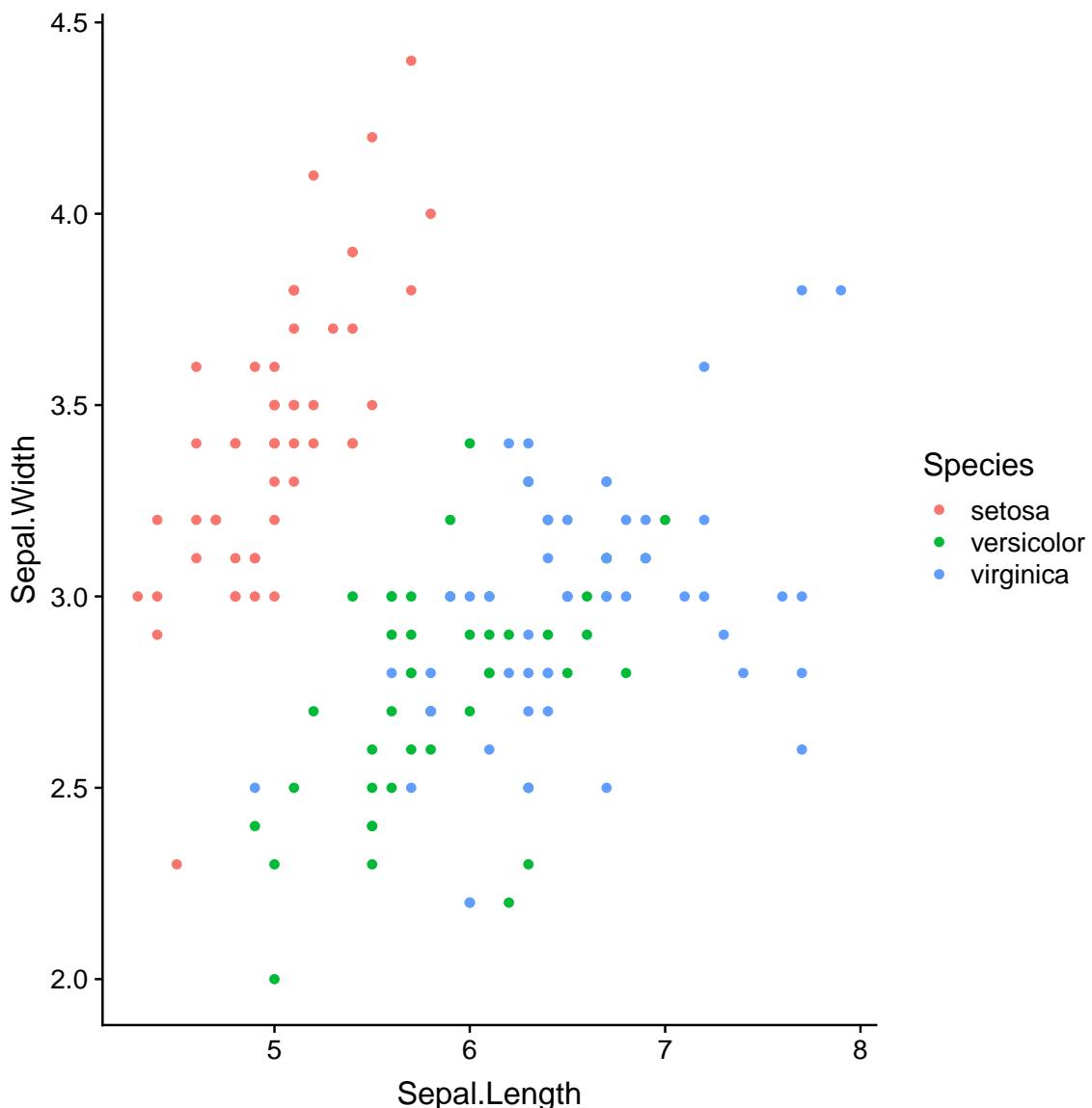
```
> data(iris)
> head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5         1.4         0.2   setosa
2          4.9         3.0         1.4         0.2   setosa
3          4.7         3.2         1.3         0.2   setosa
4          4.6         3.1         1.5         0.2   setosa
5          5.0         3.6         1.4         0.2   setosa
6          5.4         3.9         1.7         0.4   setosa

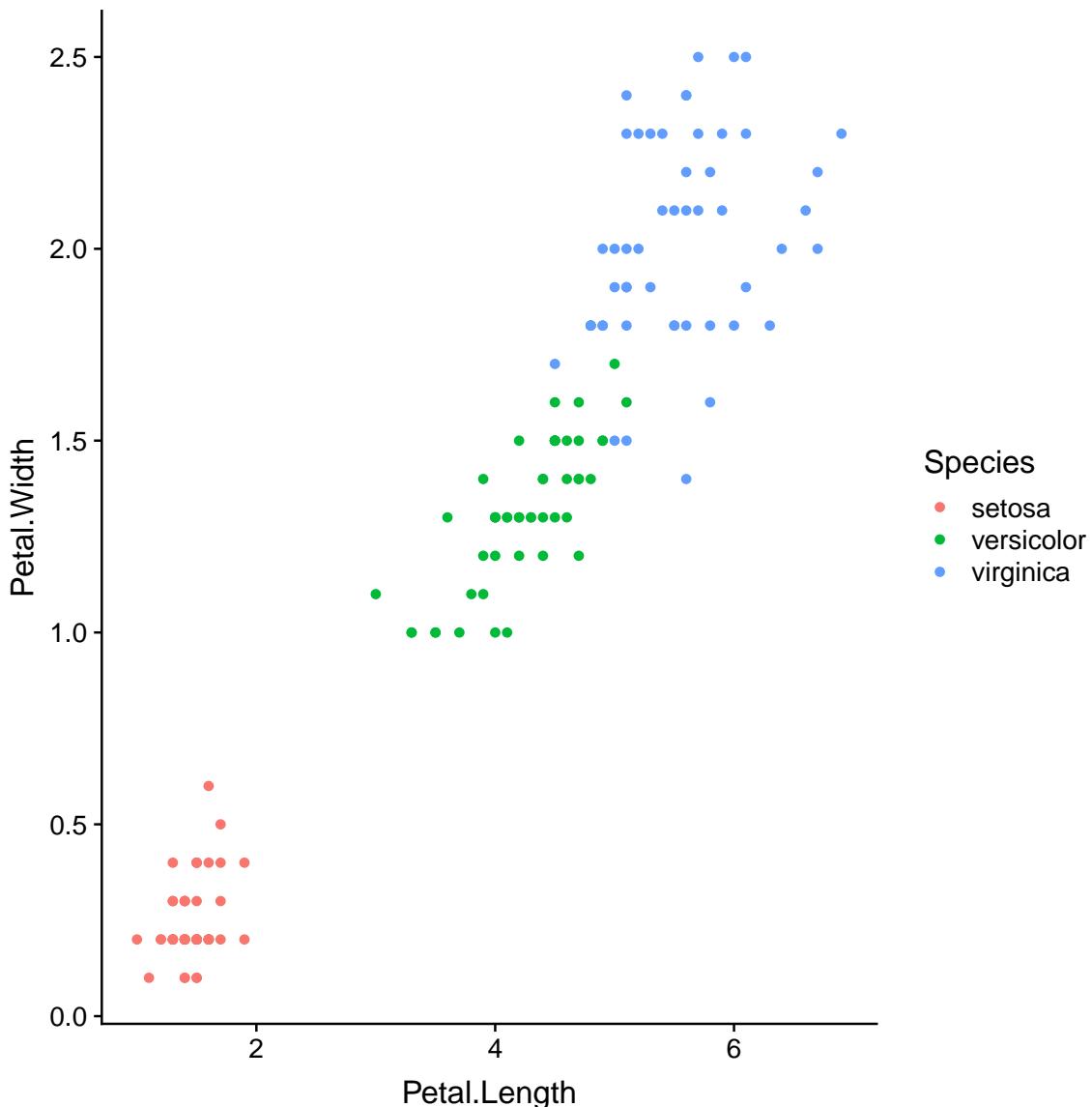
> str(iris)

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

> ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
+     geom_point(aes(color = Species))
```



```
> ggplot(data = iris, aes(x = Petal.Length, y = Petal.Width)) +  
+     geom_point(aes(color = Species))
```



Here the potential predictors are sepal width and length and petal width and length, and the goal is to find a classifier that will yield the correct species. From the scatter plots it should be pretty clear that a model with petal length and petal width as predictors would classify the data well. Although in a sense this is too easy, these data do a good job of illustrating logistic regression with more than two classes.

Before doing that we randomly choose 75 of the 150 rows of the data frame to be the training sample, with the other 75 being the test sample.

```
> set.seed(321)
> selected <- sample(1:150, replace = FALSE, size = 75)
> iris.train <- iris[selected, ]
> iris.test <- iris[-selected, ]
```

There are several packages which implement logistic regression for data with more than two classes. We will use the VGAM package. The function `vglm` within VGAM implements logistic regression (and much more).

```
> library(VGAM)
> iris.lr <- vglm(Species ~ Petal.Width + Petal.Length, data = iris.train,
+   family = multinomial)
> summary(iris.lr)

Call:
vglm(formula = Species ~ Petal.Width + Petal.Length, family = multinomial,
      data = iris.train)

Pearson residuals:
              Min           1Q         Median
log(mu[,1]/mu[,3]) -0.000001813 -0.0000002464 0.0000000436
log(mu[,2]/mu[,3]) -1.265281432 -0.0001223330 0.0000006589
                               3Q           Max
log(mu[,1]/mu[,3]) 0.00000006302 0.00001644
log(mu[,2]/mu[,3]) 0.00299288301 2.47065936

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept):1    118.947    9260.552     NA      NA
(Intercept):2      63.510     34.004     NA      NA
Petal.Width:1    -27.338   14731.810    -0.002    0.999
Petal.Width:2    -14.554      7.466     NA      NA
Petal.Length:1   -26.612    6453.201    -0.004    0.997
Petal.Length:2    -8.242      5.591     NA      NA

Number of linear predictors: 2

Names of linear predictors:
log(mu[,1]/mu[,3]), log(mu[,2]/mu[,3])

Residual deviance: 8.6819 on 144 degrees of freedom

Log-likelihood: -4.341 on 144 degrees of freedom

Number of iterations: 21

Warning: Hauck-Donner effect detected in the following estimate(s):
```

```
'(Intercept):1', '(Intercept):2', 'Petal.Width:2', 'Petal.Length:2'
```

Reference group is level 3 of the response

Notice that the family is specified as `multinomial` rather than `binomial` since we have more than two classes. When run with these data, the `vglm` function returns several (about 20) warnings. These occur mainly because the classes are so easily separated, and are suppressed above.

Next we compute the probabilities for the test data.

At least for the first six cases, one probability is close to one and the other two are close to zero, reflecting the fact that this is an easy classification problem. Next we extract the actual predictions. For these, we want to choose the highest probability in each row. The `which.max` function makes this easy. Before applying this to the fitted probabilities, we illustrate its use.

```
> which.max(c(2, 3, 1, 5, 8, 3))  
[1] 5  
  
> which.max(c(2, 20, 4, 5, 9, 1, 0))  
[1] 2  
  
> class.predictions <- apply(iris.probs, 1, which.max)  
> head(class.predictions)
```

```

4 5 9 11 12 14
1 1 1 1 1 1

> class.predictions[class.predictions == 1] <- levels(iris$Species)[1]
> class.predictions[class.predictions == 2] <- levels(iris$Species)[2]
> class.predictions[class.predictions == 3] <- levels(iris$Species)[3]
> head(class.predictions)

        4      5      9     11     12     14
"setosa" "setosa" "setosa" "setosa" "setosa" "setosa"

```

Next we create the confusion matrix.

```
> table(class.predictions, iris.test$Species)
```

| class.predictions | setosa | versicolor | virginica |
|-------------------|--------|------------|-----------|
| setosa            | 26     | 0          | 0         |
| versicolor        | 0      | 19         | 1         |
| virginica         | 0      | 2          | 27        |

## 9.2 Nearest neighbor methods

Nearest neighbor methods provide a rather different way to construct classifiers, and have strengths (minimal assumptions, flexible decision boundaries) and weaknesses (computational burden, lack of interpretability) compared to logistic regression models.

In principle the idea is simple. Recall that the training set will have both  $x$  and  $y$  values known, while the test set will have only  $x$  values known. Begin by choosing a positive integer  $k$  which will specify the number of neighbors to use in classification. To classify a point  $x$  in the training set, find the  $k$  closest  $x$  values in the training set, and choose the class which has the highest representation among the  $k$  points. The algorithm is called kNN for “ $k$  Nearest Neighbors.”

For example, suppose that  $k = 10$  and the 10 nearest neighbors to a training  $x$  have classes 1, 1, 3, 2, 3, 3, 3, 2, 3, 2. Since there are five 3s, three 2s, and two 1s, the training point is assigned to class 3. Suppose that for another  $x$  the 10 nearest neighbors have classes 1, 1, 1, 2, 3, 1, 3, 3, 3, 2. In this case there are four 1s and four 3s, so there is a tie for the lead. The nearest neighbor algorithm then will choose between 1 and 3 at random.

Although in principle kNN is simple, some issues arise. First, how should  $k$  be chosen? There is not an easy answer, but it can help to think of the extreme values for  $k$ .

The largest possible  $k$  is the number of observations in the training set. For example suppose that the training set has 10 observations, with classes 1, 1, 1, 2, 2, 2, 3, 3, 3, 3. Then for any point in the test set, the  $k = 10$  nearest neighbors will include ALL of the points in the training set, and hence every point in the test set will be classified in class 3. This classifier has low (zero) variance, but probably has high bias.

The smallest possible  $k$  is 1. In this case, each point in the test set is put in the same class as its nearest neighbor in the training set. This may lead to a very non-smooth and high variance classifier, but the bias will tend to be small.

A second issue that is relatively easy to deal with concerns the scales on which the  $x$  values are measured. If for example one  $x$  variable has a range from 2 to 4, while another has a range from 2000 to 4000, the distance between the test and training data will be dominated by the second variable. The solution that is typically used is to standardize all the variables (rescale them so that their mean is 0 and their standard deviation 1).

These and other issues are discussed in the literature on kNN, but we won't pursue them further.

There are at least three R packages which implement kNN, including `class`, `kknn`, and `RWeka`. We will use `class` below.

An example from [Hastie et al. \[2009\]](#) will be used to give a better sense of the role of  $k$  in the kNN algorithm. The example uses simulated data and shows the decision boundaries for kNN with  $k = 15$  and  $k = 1$ .<sup>1</sup> Although the R code used to draw the displays is given below, focus mainly on the graphics produced, and what they tell us about kNN.

First the data are read into R and graphed. The predictors `x1` and `x2`, while not standardized, have very similar standard deviations, so we will not standardize these data before applying kNN.

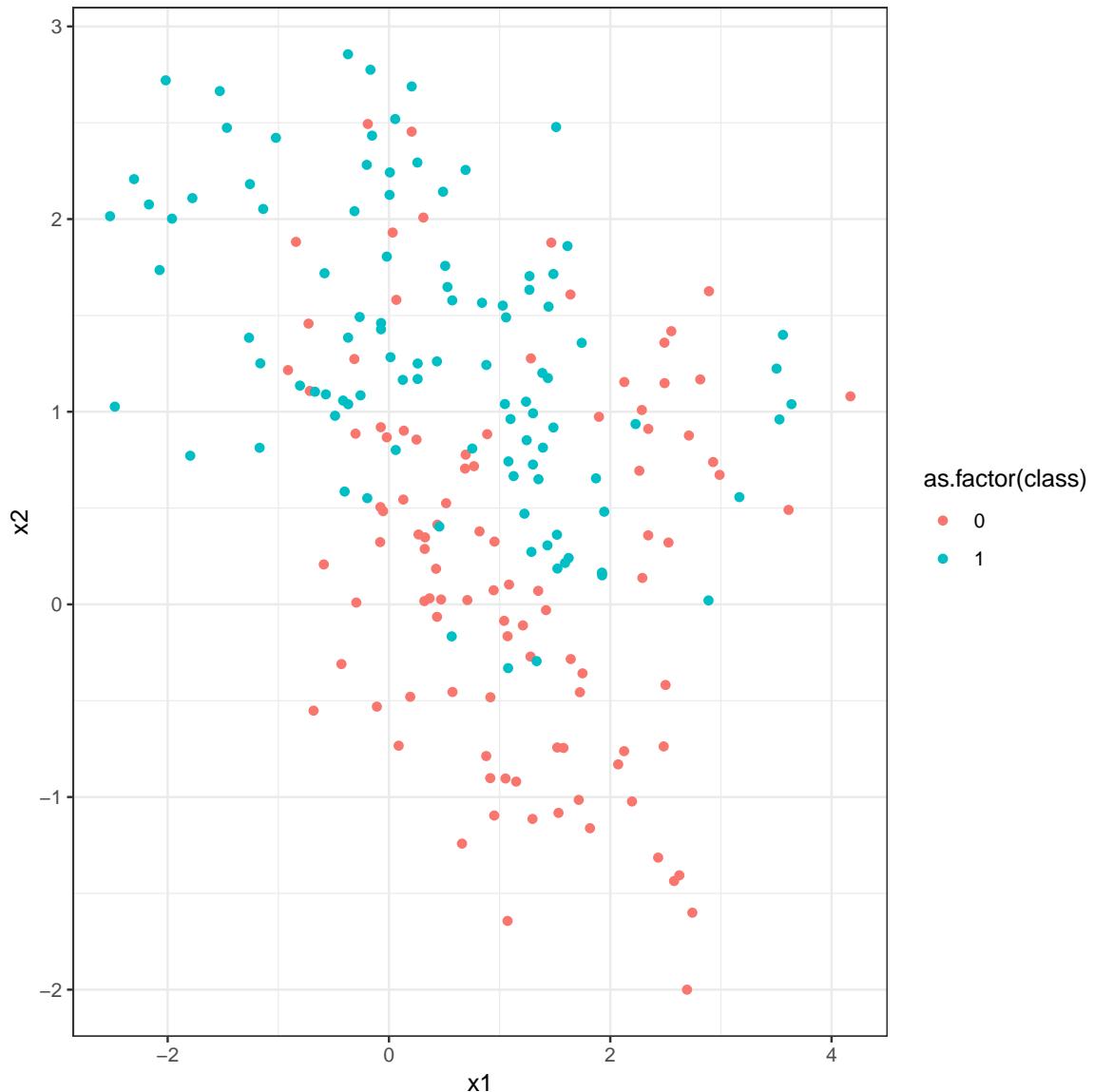
```
> u.knn <- "http://blue.for.msu.edu/FOR875/data/knnExample.csv"
> knnExample <- read.csv(u.knn, header=TRUE)
> str(knnExample)

'data.frame': 200 obs. of  3 variables:
 $ x1    : num  2.5261 0.367 0.7682 0.6934 -0.0198 ...
 $ x2    : num  0.3211 0.0315 0.7175 0.7772 0.8673 ...
 $ class: int  0 0 0 0 0 0 0 0 0 0 ...


> ggplot(data = knnExample, aes(x = x1, y = x2)) +
+   geom_point(aes(color = as.factor(class))) +
+   theme_bw()
```

---

<sup>1</sup>The graphs below use code adapted from <http://stackoverflow.com/questions/31234621/variation-on-how-to-plot-decision-boundary-of-a-k-nearest-neighbor-classifier-f>



Next a large set of test data is created using the `expand.grid` function, which creates a data frame with all possible combinations of the arguments. First a simple example to illustrate the function, then the actual creation of the training set. The test set covers the range of the `x1` and `x2` values in the training set.

```
> expand.grid(x = c(1, 2), y = c(5, 3.4, 2))
```

|   | x | y   |
|---|---|-----|
| 1 | 1 | 5.0 |
| 2 | 2 | 5.0 |
| 3 | 1 | 3.4 |
| 4 | 2 | 3.4 |
| 5 | 1 | 2.0 |

```

6 2 2.0

> min(knnExample$x1)
[1] -2.52082

> max(knnExample$x1)
[1] 4.170746

> min(knnExample$x2)
[1] -1.999853

> max(knnExample$x2)
[1] 2.855805

> x.test <- expand.grid(x1 = seq(-2.6, 4.2, by = 0.1), x2 = seq(-2,
+      2.9, by = 0.1))

```

Next the kNN with  $k = 15$  is fit. Notice that the first argument gives the  $x$  values in the training set, the next argument gives the  $x$  values in the test set, the third argument gives the  $y$  values (labels) from the training set. The fourth argument gives  $k$ , and the fifth argument asks for the function to return the probabilities of membership (that is, the proportion of the nearest neighbors which were in the majority class) as well as the class assignments.

```

> library(class)
> Example_knn <- knn(knnExample[, c(1, 2)], x.test, knnExample[, 
+      3], k = 15, prob = TRUE)
> prob <- attr(Example_knn, "prob")
> head(prob)

[1] 0.6666667 0.6666667 0.6666667 0.7333333 0.7333333
[6] 0.7333333

```

Next the graphs are created. This is somewhat complex, since we want to plot the test data colored by the class they were assigned to by the kNN classifier as background, the training data (using a different symbol), and the decision boundary.

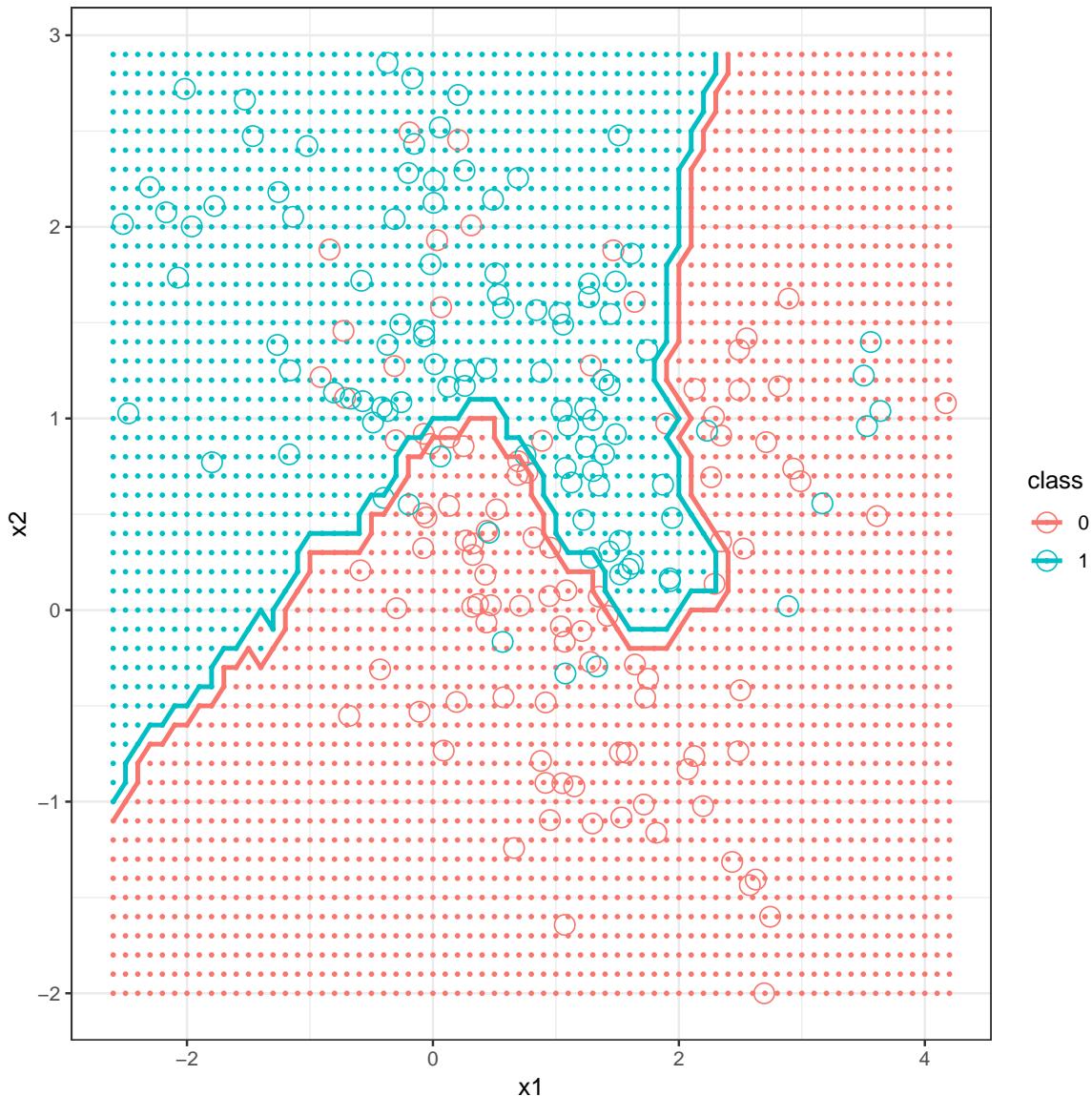
```
> library(dplyr)
> df1 <- mutate(x.test, prob = prob, class = 0, prob_cls = ifelse(Example_knn ==
+     class, 1, 0))
> str(df1)

'data.frame': 3450 obs. of  5 variables:
$ x1      : num  -2.6 -2.5 -2.4 -2.3 -2.2 ...
$ x2      : num  -2 -2 -2 -2 -2 ...
$ prob    : num  0.667 0.667 0.667 0.733 0.733 ...
$ class   : num  0 0 0 0 0 0 0 0 0 ...
$ prob_cls: num  1 1 1 1 1 1 1 1 1 ...

> df2 <- mutate(x.test, prob = prob, class = 1, prob_cls = ifelse(Example_knn ==
+     class, 1, 0))
> bigdf <- bind_rows(df1, df2)
>
> names(knnExample)

[1] "x1"      "x2"      "class"

> ggplot(bigdf) + geom_point(aes(x = x1, y = x2, col = class),
+     data = mutate(x.test, class = Example_knn), size = 0.5) +
+     geom_point(aes(x = x1, y = x2, col = as.factor(class)), size = 4,
+     shape = 1, data = knnExample) + geom_contour(aes(x = x1,
+     y = x2, z = prob_cls, group = as.factor(class), color = as.factor(class)),
+     size = 1, bins = 1, data = bigdf) + theme_bw()
```



Next we graph the decision boundary of kNN with  $k = 1$ .

```
> Example_knn <- knn(knnExample[, c(1, 2)], x.test, knnExample[,
+      3], k = 1, prob = TRUE)
> prob <- attr(Example_knn, "prob")
> head(prob)

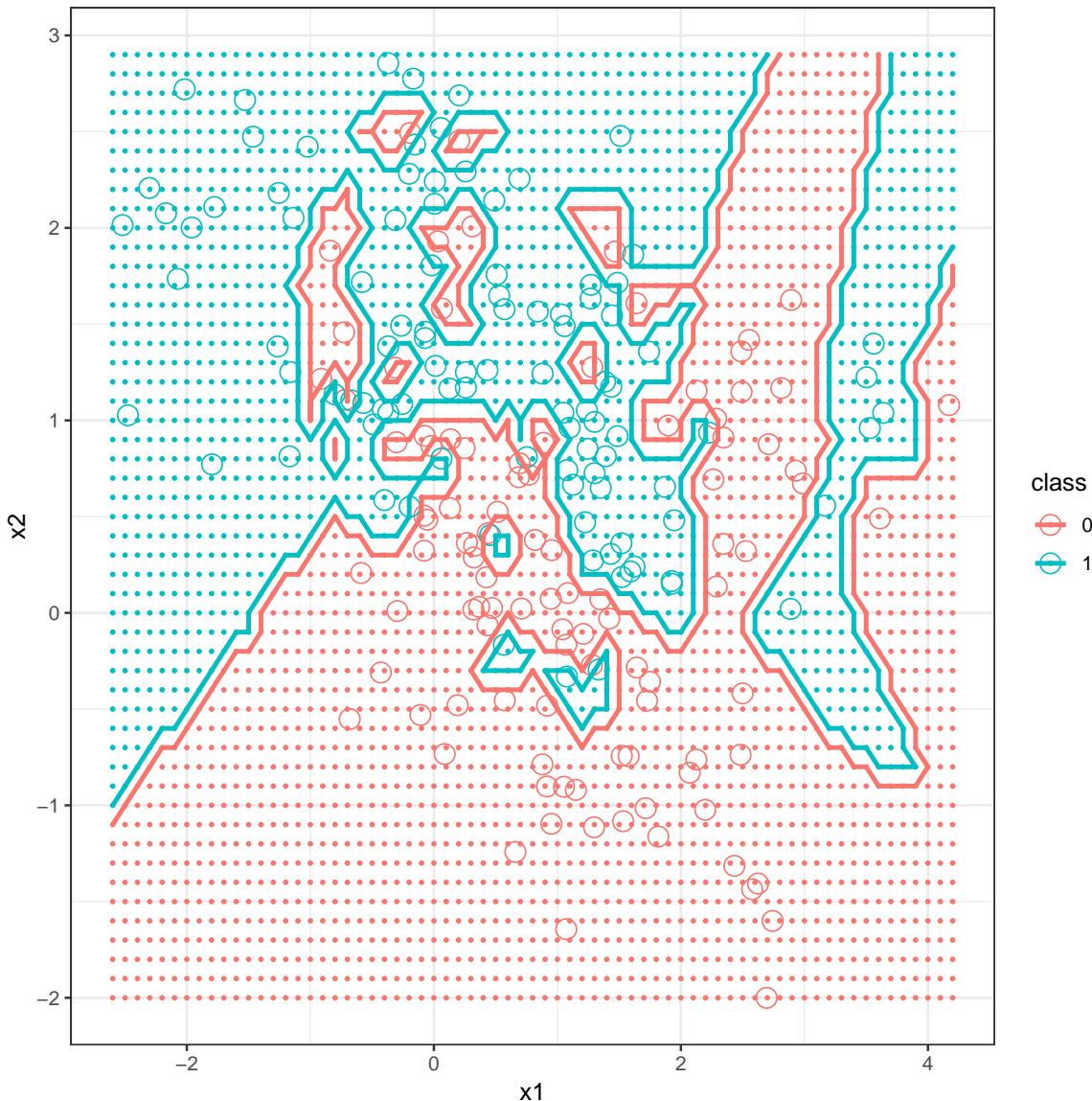
[1] 1 1 1 1 1 1

> df1 <- mutate(x.test, prob = prob, class = 0, prob_cls = ifelse(Example_knn ==
+      class, 1, 0))
> str(df1)

'data.frame': 3450 obs. of  5 variables:
```

```
$ x1      : num  -2.6 -2.5 -2.4 -2.3 -2.2 -2.1 -2 -1.9 -1.8 -1.7 ...
$ x2      : num  -2 -2 -2 -2 -2 -2 -2 -2 -2 ...
$ prob    : num  1 1 1 1 1 1 1 1 1 1 ...
$ class   : num  0 0 0 0 0 0 0 0 0 0 ...
$ prob_cls: num  1 1 1 1 1 1 1 1 1 1 ...

> df2 <- mutate(x.test, prob = prob, class = 1, prob_cls = ifelse(Example_knn ==
+     class, 1, 0))
> bigdf <- bind_rows(df1, df2)
>
> ggplot(bigdf) + geom_point(aes(x = x1, y = x2, col = class),
+     data = mutate(x.test, class = Example_knn), size = 0.5) +
+     geom_point(aes(x = x1, y = x2, col = as.factor(class)), size = 4,
+     shape = 1, data = knnExample) + geom_contour(aes(x = x1,
+     y = x2, z = prob_cls, group = as.factor(class), color = as.factor(class)),
+     size = 1, bins = 1, data = bigdf) + theme_bw()
```



### 9.2.1 kNN and the diabetes data

Next kNN is applied to the diabetes data. We will use the same predictors, `glu` and `bmi` that were used in the logistic regression example. Since the scales of the predictor variables are substantially different, they are standardized first. The value  $k = 15$  is chosen for kNN.

```
> Pima.tr[, 1:7] <- scale(Pima.tr[, 1:7], center = TRUE, scale = TRUE)
> Pima.te[, 1:7] <- scale(Pima.te[, 1:7], center = TRUE, scale = TRUE)
> knn_Pima <- knn(Pima.tr[, c(2, 5)], Pima.te[, c(2, 5)], Pima.tr[, ,
+     8], k = 15, prob = TRUE)
> table(knn_Pima, Pima.te[, 8])
```

| knn_Pima | No  | Yes |
|----------|-----|-----|
| No       | 206 | 55  |
| Yes      | 17  | 54  |

At least in terms of the confusion matrix, kNN with  $k = 15$  performed about the same as logistic regression for these data.

### 9.2.2 kNN and the iris data

Now kNN is used to classify the iris data. As before we use petal length and width as predictors. The scales of the two predictors are not particularly different, so we won't standardize the predictors. Unsurprisingly kNN does well in classifying the test set for a wide variety of  $k$  values.

```
> sd(iris.train$Petal.Width)
[1] 0.728585

> sd(iris.train$Petal.Length)
[1] 1.671873

> head(iris.train)

  Sepal.Length Sepal.Width Petal.Length Petal.Width
144          6.8        3.2       5.9        2.3
140          6.9        3.1       5.4        2.1
36           5.0        3.2       1.2        0.2
38           4.9        3.6       1.4        0.1
58           4.9        2.4       3.3        1.0
50           5.0        3.3       1.4        0.2
  Species
144 virginica
140 virginica
36   setosa
38   setosa
58 versicolor
50   setosa

> knn_iris <- knn(iris.train[, c(3, 4)], iris.test[, c(3, 4)],
+                   iris.train[, 5], k = 1, prob = TRUE)
> table(knn_iris, iris.test[, 5])
```

```
knn_iris      setosa versicolor virginica
setosa        26         0         0
versicolor     0        20         1
virginica      0         1        27

> knn_iris <- knn(iris.train[, c(3, 4)], iris.test[, c(3, 4)],
+       iris.train[, 5], k = 3, prob = TRUE)
> table(knn_iris, iris.test[, 5])

knn_iris      setosa versicolor virginica
setosa        26         0         0
versicolor     0        19         1
virginica      0         2        27

> knn_iris <- knn(iris.train[, c(3, 4)], iris.test[, c(3, 4)],
+       iris.train[, 5], k = 15, prob = TRUE)
> table(knn_iris, iris.test[, 5])

knn_iris      setosa versicolor virginica
setosa        26         0         0
versicolor     0        19         1
virginica      0         2        27
```

**Exercise 12**

Learning objectives: explore the logistic regression classification method; apply the kNN classification method; create confusion matrices to compare classification methods; plot classified data.

# Chapter 10

## Text Data

Many applications require the ability to manipulate and process text data. For example, an email spam filter takes as its input various features of email such as the sender, words in the subject, words in the body, the number and types of attachments, and so on, and the filter tries to build a classifier which can correctly classify a message as spam or not spam. As another example, some works of literature such as some of Shakespeare's plays or some of the Federalist papers have disputed authorship. By analyzing word use across many documents, researchers try to determine the author of the disputed work.

Working with text data requires functions that will, for example, concatenate and split text strings, modify strings (e.g., converting to lower-case or removing vowels), count the number of characters in a string, and so on. In addition to being useful in such contexts, string manipulation is helpful more generally in R—for example, to effectively construct titles for graphics.

As with most tasks, there are a variety of ways to accomplish these text processing tasks in R. The base R package has functions which work with and modify text strings. Another useful package which approaches these tasks in a slightly different way is `stringr`. As with graphics, we will focus mainly on one package, to avoid confusion. In this case we will focus on the base R string processing functions, but will emphasize that `stringr` is also worth knowing.

The application to analyzing *Moby Dick* below comes from the book *Text Analysis with R for Students of Literature* by Matthew L. Jockers.

### 10.1 Reading text data into R

Often text data will not be in a rectangular format, suitable for reading into a data frame. For example, an email that would be used to help train a spam filter, or literary texts that could be used to help determine authorship of a novel are certainly not of this form. Often when working with text data we want to read the whole text object into a single R vector. In this case either the `scan` function or the `readLines` function

are useful. The `readLines` function is typically more efficient, but `scan` is much more flexible.

As an example, consider the email file in Figure 10.1 and a plain text version of the novel *Moby Dick* by Herman Melville, the beginning of which is displayed in Figure 10.2. The email message is available at <http://blue.for.msu.edu/FOR875/data/email.txt> while the novel is available at <http://blue.for.msu.edu/FOR875/data/mobydick.txt>. We will read these into R using `scan`.

```

From safety33o@111.newnamedns.com  Fri Aug 23 11:03:37 2002
Return-Path: <safety33o@111.newnamedns.com>
Delivered-To: zzzz@localhost.example.com
Received: from localhost (localhost [127.0.0.1])
    by phobos.labs.example.com (Postfix) with ESMTP id 5AC994415F
    for <zzzz@localhost>; Fri, 23 Aug 2002 06:02:59 -0400 (EDT)
Received: from mail.webnote.net [193.120.211.219]
    by localhost with POP3 (fetchmail-5.9.0)
    for zzzz@localhost (single-drop); Fri, 23 Aug 2002 11:02:59 +0100 (IST)
Received: from 111.newnamedns.com ([64.25.38.81])
    by webnote.net (8.9.3/8.9.3) with ESMTP id KAA09379
    for <zzzz@example.com>; Fri, 23 Aug 2002 10:18:03 +0100
From: safety33o@111.newnamedns.com
Date: Fri, 23 Aug 2002 02:16:25 -0400
Message-Id: <200208230616.g7N6GOR28438@111.newnamedns.com>
To: kxzzzgxlrah@111.newnamedns.com
Reply-To: safety33o@111.newnamedns.com
Subject: ADV: Lowest life insurance rates available!
moodie

Lowest rates available for term life insurance! Take a moment
and fill out our online form
to see the low rate you qualify for.
Save up to 70% from regular rates! Smokers accepted!
http://www.newnamedns.com/termlife/

Representing quality nationwide carriers. Act now!
```

Figure 10.1: A spam email message

First, we read in the email message. The `scan` function has several possible arguments. For now the important arguments are the file to be read (the argument is named `file`), the type of data in the file (the argument is named `what`), and how the fields in the file are separated (the argument is named `sep`). To illustrate the `sep` argument, the file will be read into R once with `sep = ""` indicating that the separator is whitespace, and once with `sep = "\n"` indicating that the separator is the newline character, i.e., each field in the file is a line.

The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman Melville

This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at [www.gutenberg.org](http://www.gutenberg.org)

Title: Moby Dick; or The Whale

Author: Herman Melville

Last Updated: January 3, 2009

Posting Date: December 25, 2008 [EBook #2701]

Release Date: June, 2001

Language: English

\*\*\* START OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE \*\*\*

Produced by Daniel Lazarus and Jonesey

MOBY DICK; OR THE WHALE

By Herman Melville

Figure 10.2: The beginning of a plain text file from Project Gutenberg containing the novel *Moby Dick*

```

> u.email <- "http://blue.for.msu.edu/FOR875/data/email1.txt"
> email1 <- scan(u.email, what = "character", sep = "")
> length(email1)

[1] 133

> email1[1:10]

[1] "From"
[2] "safety33o@l11.newnamedns.com"
[3] "Fri"
[4] "Aug"
[5] "23"
[6] "11:03:37"
[7] "2002"
[8] "Return-Path:"
[9] "<safety33o@l11.newnamedns.com>"
[10] "Delivered-To:"

> email1 <- scan(u.email, what = "character", sep = "\n")
> length(email1)

[1] 26

> email1[1:10]

[1] "From safety33o@l11.newnamedns.com Fri Aug 23 11:03:37 2002"
[2] "Return-Path: <safety33o@l11.newnamedns.com>"
[3] "Delivered-To: zzzz@localhost.example.com"
[4] "Received: from localhost (localhost [127.0.0.1])"
[5] "\tby phobos.labs.example.com (Postfix) with ESMTP id 5AC994415F"
[6] "\tfor <zzzz@localhost>; Fri, 23 Aug 2002 06:02:59 -0400 (EDT)"
[7] "Received: from mail.webnote.net [193.120.211.219]"
[8] "\tby localhost with POP3 (fetchmail-5.9.0)"
[9] "\tfor zzzz@localhost (single-drop); Fri, 23 Aug 2002 11:02:59 +0100 (IST)"
[10] "Received: from l11.newnamedns.com ([64.25.38.81])"

```

Note that when `sep = ""` was specified, every time whitespace was encountered R moved to a new element of the vector `email1`, and this vector ultimately contained 133 elements. When `sep = "\n"` was specified, all the text before a newline was put into one element of the vector, which ended up with 26 elements.

The `scan` function is quite flexible. In fact, `read.table` uses `scan` to actually read in the data. Read the help file for `scan` if more information is desired.

Next *Moby Dick* is read in line by line.

```
> u.moby <- "http://blue.for.msu.edu/FOR875/data/mobydick.txt"
> moby_dick <- scan(u.moby, what = "character", sep = "\n")
> moby_dick[1:25]

[1] "The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman Melville"
[2] "This eBook is for the use of anyone anywhere at no cost and with"
[3] "almost no restrictions whatsoever. You may copy it, give it away or"
[4] "re-use it under the terms of the Project Gutenberg License included"
[5] "with this eBook or online at www.gutenberg.org"
[6] "Title: Moby Dick; or The Whale"
[7] "Author: Herman Melville"
[8] "Last Updated: January 3, 2009"
[9] "Posting Date: December 25, 2008 [EBook #2701]"
[10] "Release Date: June, 2001"
[11] "Language: English"
[12] "*** START OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE ***"
[13] "Produced by Daniel Lazarus and Jonesey"
[14] "MOBY DICK; OR THE WHALE"
[15] "By Herman Melville"
[16] "Original Transcriber's Notes:"
[17] "This text is a combination of etexts, one from the now-defunct ERIS"
[18] "project at Virginia Tech and one from Project Gutenberg's archives. The"
[19] "proofreaders of this version are indebted to The University of Adelaide"
[20] "Library for preserving the Virginia Tech version. The resulting etext"
[21] "was compared with a public domain hard copy version of the text."
[22] "In chapters 24, 89, and 90, we substituted a capital L for the symbol"
[23] "for the British pound, a unit of currency."
[24] "ETYMOLOGY."
[25] "(Supplied by a Late Consumptive Usher to a Grammar School)"
```

You will notice that `scan` function ignored blank lines in the file. If it is important to preserve blank lines, the argument `blank.lines.skip = FALSE` can be supplied to `scan`.

The file containing the novel contains some introductory and closing text that is not part of the original novel. If we are interested in Melville's writing, we should remove this text. By inspection we can discover that the novel's text begins at position 408 and ends at position 18576.

```
> moby_dick <- moby_dick[408:18576]
> length(moby_dick)

[1] 18169
```

```
> moby_dick[1:4]

[1] "CHAPTER 1. Loomings."
[2] "Call me Ishmael. Some years ago--never mind how long precisely--having"
[3] "little or no money in my purse, and nothing particular to interest me on"
[4] "shore, I thought I would sail about a little and see the watery part of"

> moby_dick[18165:18169]

[1] "they glided by as if with padlocks on their mouths; the savage sea-hawks"
[2] "sailed with sheathed beaks. On the second day, a sail drew near, nearer,"
[3] "and picked me up at last. It was the devious-cruising Rachel, that in"
[4] "her retracing search after her missing children, only found another"
[5] "orphan."
```

## 10.2 The paste function

The `paste` function concatenates vectors after (if necessary) converting the vectors to character.

```
> paste("Homer Simpson", "is", "Bart Simpson's", "father")

[1] "Homer Simpson is Bart Simpson's father"

> n <- 10
> paste("The value of n is", n)

[1] "The value of n is 10"

> paste(c("pig", "dog"), 3)

[1] "pig 3" "dog 3"
```

By default the `paste` function separates the input vectors with a space. But other separators can be specified.

```
> paste("mail", "google", "com", sep = ".")
[1] "mail.google.com"

> paste("and", "or", sep = "/")
```

```
[1] "and/or"

> paste(c("dog", "cat", "horse", "human", "elephant"), "food")

[1] "dog food"      "cat food"      "horse food"
[4] "human food"    "elephant food"
```

Sometimes we want to take a character vector with  $n$  elements and create a character vector with only one element, which contains all  $n$  character strings. Setting the `collapse` argument to something other than the default `NULL` tells R we want to do this, and allows specification of the separator in the collapsed vector.

```
> paste(c("one", "two", "three", "four", "five"), c("six", "seven",
+       "eight", "nine", "ten"))

[1] "one six"      "two seven"     "three eight"   "four nine"
[5] "five ten"

> paste(c("one", "two", "three", "four", "five"), c("six", "seven",
+       "eight", "nine", "ten"), collapse = ".") 

[1] "one six.two seven.three eight.four nine.five ten"

> paste(c("one", "two", "three", "four", "five"), c("six", "seven",
+       "eight", "nine", "ten"), collapse = "&&") 

[1] "one six&&two seven&&three eight&&four nine&&five ten"

> paste(c("one", "two", "three", "four", "five"), c("six", "seven",
+       "eight", "nine", "ten"), collapse = " ") 

[1] "one six two seven three eight four nine five ten"
```

In the example above by default `paste` created a vector with five elements, each containing one input string from the first input vector and one from the second vector, pasted together. When a non `NULL` argument was specified for `collapse`, the vector created had one element, with the pasted strings separated by that argument.<sup>1</sup>

Also don't forget that R "recycles" values from vectors if two or more different length vectors are provided as input.

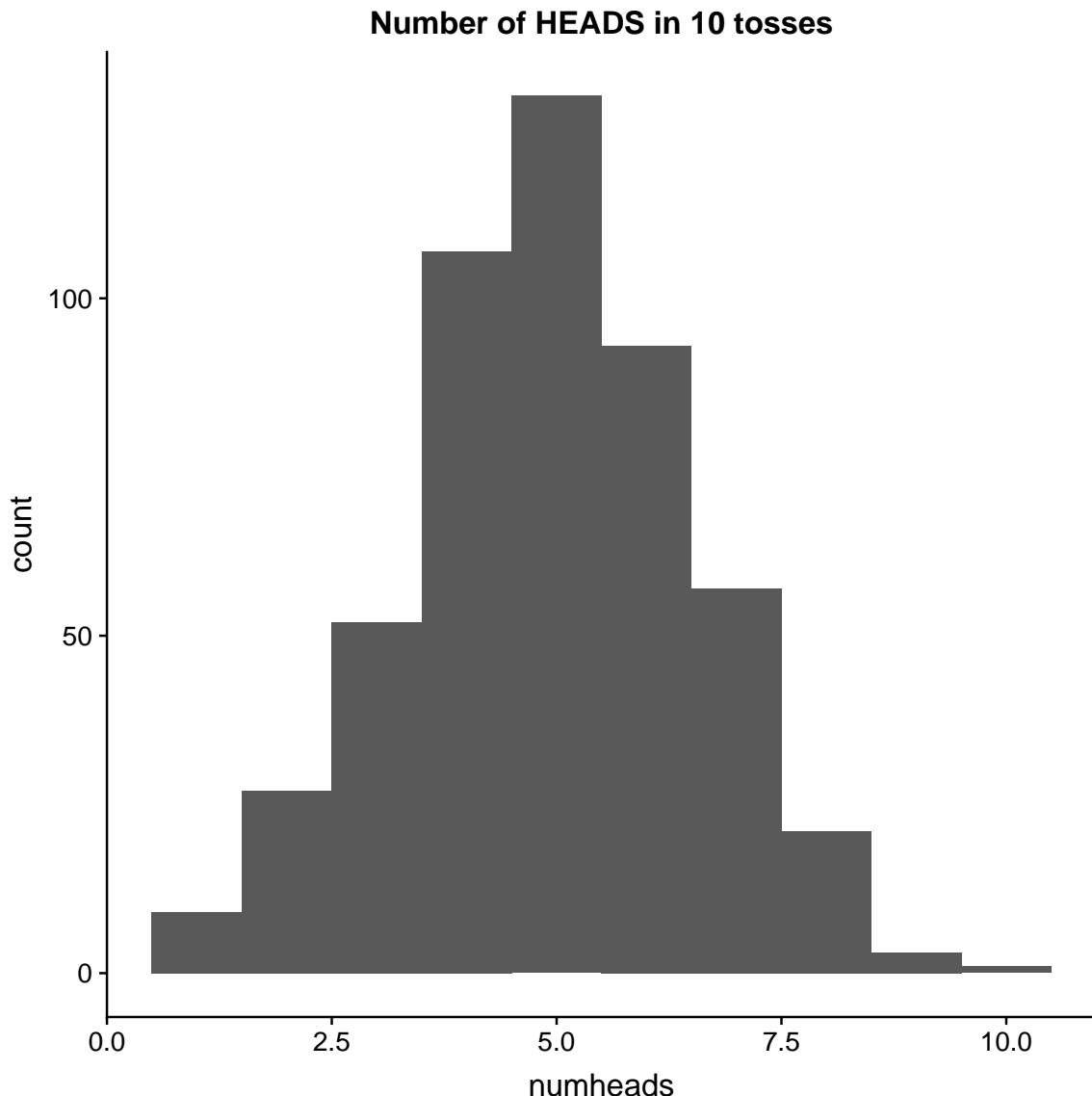
---

<sup>1</sup>There is a somewhat subtle difference among the examples. If all the arguments are length one vectors, then `paste` by default returns a length one vector. But if one or more of the arguments have length greater than one, the default behavior of `paste` is to return a vector of length greater than one. The `collapse` argument changes this behavior.

```
> paste(c("a", "b"), 1:10, sep = "")  
  
[1] "a1"  "b2"  "a3"  "b4"  "a5"  "b6"  "a7"  "b8"  "a9"  
[10] "b10"  
  
> paste(c("a", "b"), 1:9, sep = "")  
  
[1] "a1"  "b2"  "a3"  "b4"  "a5"  "b6"  "a7"  "b8"  "a9"
```

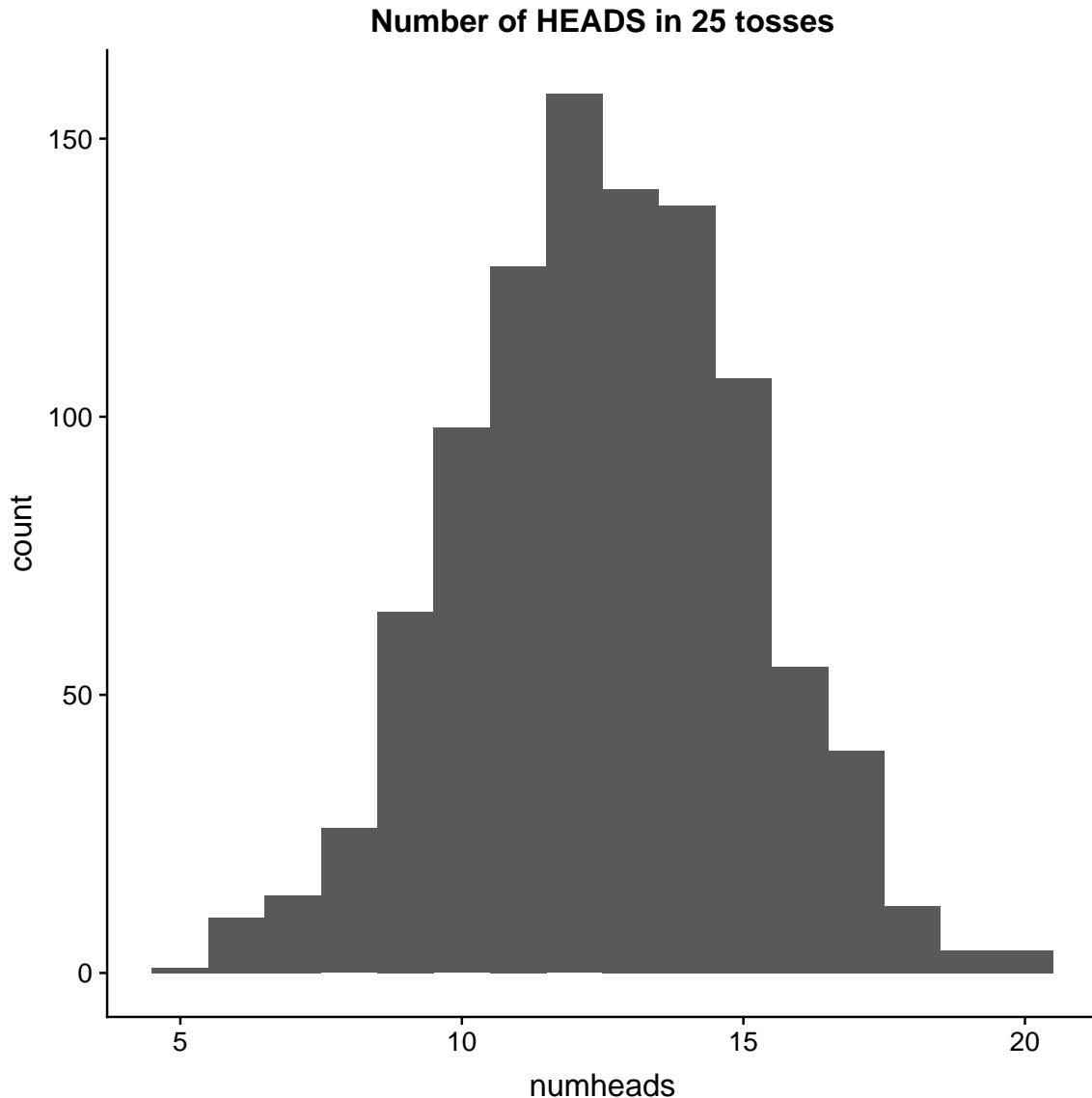
Next, consider writing a function which simulates repeatedly tossing a coin  $n$  times, counting the number of HEADS out of the  $n$  tosses. For the first five repetitions of  $n$  tosses, the function will print out the number of HEADS (for example if there are 7 HEADS in the  $n = 10$  tosses the function prints "The number of HEADS out of 10 tosses is 7." The function returns a histogram of the number of HEADS, with a title stating "Number of HEADS in ?? tosses" where ?? is replaced by the number of tosses. The `paste` function will help greatly.

```
> coin_toss <- function(n = 10, iter = 500) {  
+   require(ggplot2)  
+   df <- data.frame(numheads = numeric(iter))  
+   for (i in 1:iter) {  
+     df$numheads[i] <- rbinom(1, n, 0.5)  
+     if (i <= 5) {  
+       print(paste("The number of HEADS out of", n, "tosses is",  
+                   df$numheads[i]))  
+     }  
+   }  
+   ggplot(data = df, aes(x = numheads)) + geom_histogram(binwidth = 1) +  
+     ggtitle(paste("Number of HEADS in", n, "tosses"))  
+ }  
> coin_toss()  
  
[1] "The number of HEADS out of 10 tosses is 7"  
[1] "The number of HEADS out of 10 tosses is 5"  
[1] "The number of HEADS out of 10 tosses is 7"  
[1] "The number of HEADS out of 10 tosses is 6"  
[1] "The number of HEADS out of 10 tosses is 5"
```



```
> coin_toss(n = 25, iter = 1000)

[1] "The number of HEADS out of 25 tosses is 12"
[1] "The number of HEADS out of 25 tosses is 13"
[1] "The number of HEADS out of 25 tosses is 19"
[1] "The number of HEADS out of 25 tosses is 11"
[1] "The number of HEADS out of 25 tosses is 17"
```



Returning to the object `moby_dick` which contains the text of the novel, if we want to analyze word choice, word frequency, etc., it would be helpful to form a vector in which each element is a word from the novel. One way to do this is to first paste the current version of the `moby_dick`, whose elements are lines from the text file, into a new version which is one long vector with the lines pasted together. To illustrate, we will first do this with a much smaller object that shares the structure of `moby_dick`.

```
> small_novel <- c("First line", "Second somewhat longer line",
+      "third line.")
> small_novel

[1] "First line"
[2] "Second somewhat longer line"
```

```
[3] "third line."  
  
> small_novel <- paste(small_novel, collapse = " ")  
> length(small_novel)  
  
[1] 1  
  
> small_novel  
  
[1] "First line Second somewhat longer line third line."
```

Now we do the same with the actual novel.

```
> moby_dick <- paste(moby_dick, collapse = " ")  
> length(moby_dick)  
  
[1] 1
```

At this point `moby_dick` contains a single very long character string. Next we will separate this string into separate words and clean up the resulting vector a bit.

## 10.3 More string processing functions

Common string processing tasks include changing case between upper and lower, extracting and/or replacing substrings of a string, trimming a string to a specified width, counting the number of characters in a string, etc.

### `tolower` and `toupper`

R contains functions `tolower` and `toupper` which very simply change the case of all characters in a string.

```
> x <- "aBCdefG12#"  
> y <- x  
> tolower(x)  
  
[1] "abcdefg12#"  
  
> toupper(y)  
  
[1] "ABCDEFGHI12#"
```

If we are interested in frequencies of words in *Moby Dick*, converting all the text to the same case makes sense, so for example the word “the” at the beginning of a sentence is not counted differently than the same word in the middle of a sentence.

```
> moby_dick <- tolower(moby_dick)
```

## nchar and strsplit

The function **nchar** counts the number of characters in a string or strings.

```
> nchar("dog")
[1] 3

> nchar(c("dog", "cat", "horse", "elephant"))
[1] 3 3 5 8

> nchar(c("dog", "cat", "horse", "elephant", NA, "goat"))
[1] 3 3 5 8 NA 4

> nchar(c("dog", "cat", "horse", "elephant", NA, "goat"), keepNA = TRUE)
[1] 3 3 5 8 NA 4

> nchar(moby_dick)
[1] 1190309
```

By default **nchar** returns 2 for a missing value, which is a bit odd.<sup>2</sup>

The function **strsplit** splits the elements of a character vector. The function returns a list, and often the **unlist** function is useful to convert the list into an atomic vector.

```
> strsplit(c("mail.msu.edu", "mail.google.com", "www.amazon.com"),
+           split = ".", fixed = TRUE)
[[1]]
```

---

<sup>2</sup>It may be reasonable if the purpose of counting characters is to find out how much space to allocate for printing a vector of strings where the NA string will be printed.

```
[1] "mail" "msu" "edu"
[[2]]
[1] "mail" "google" "com"
[[3]]
[1] "www" "amazon" "com"

> unlist(strsplit(c("mail.msu.edu", "mail.google.com", "www.amazon.com"),
+     split = ".", fixed = TRUE))
[1] "mail" "msu" "edu" "mail" "google" "com"
[7] "www" "amazon" "com"

> unlist(strsplit(c("dog", "cat", "pig", "horse"), split = "o",
+     fixed = TRUE))
[1] "d" "g" "cat" "pig" "h" "rse"
```

Setting the argument `fixed` to `TRUE` tells R to match the value of `split` exactly when performing the split. The function can be much more powerful if the value of `split` is a *regular expression*, which can for example ask for splits at any vowels, etc. Regular expressions will be discussed more carefully in the next chapter, but we will make some use of regular expressions on a case-by-case basis prior to that.

```
> unlist(strsplit(c("dog", "cat", "pig", "horse", "rabbit"), split = "[aeiou]"))
[1] "d" "g" "c" "t" "p" "g" "h" "rs" "r" "bb" "t"
```

The regular expression `[aeiou]` represents any of the letters a, e, i, o, u. In general a string of characters enclosed in square brackets indicates any one character in the string.

```
> unlist(strsplit(c("dog", "cat", "pig", "horse", "rabbit"), split = "[aorb]"))
[1] "d" "g" "c" "t" "pig" "h" "" "se" ""
[10] "" "" "" "it"
```

The regular expression `[aorb]` represents any of the letters a, o, r, b.

```
> unlist(strsplit(c("a1c2b", "bbb2bc3f"), split = "[1-9]"))
[1] "a" "c" "b" "bbb" "bc" "f"
```

The regular expression [1-9] represents any of the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9.

```
> unlist(strsplit(c("aBc1fGh", "1TyzaaG"), split = "[^a-z"]))
[1] "a"     "c"     "f"     "h"     ""      ""      "yzaa"
```

The regular expression [a-z] represents any lower case letter. The caret ^ in front of a-z indicates “match any character *except* those in the following string” which in this case indicates “match any character that is NOT a lower case letter.”

Recall that the `moby_dick` vector now contains one long character string which includes the entire text of the novel, and that we would like to split it into separate words. We now know how to do this using `strsplit` and a regular expression. First a smaller example.

```
> unlist(strsplit(c("the rain", "in Spain stays mainly", "in",
+      "the plain"), split = "[^0-9A-Za-z"]))
[1] "the"    "rain"   "in"     "Spain"  "stays"  "mainly"
[7] "in"     "the"    "plain"

> unlist(strsplit(c("the rain", "in Spain stays mainly", "in",
+      "the plain"), split = " ", fixed = TRUE))
[1] "the"    "rain"   "in"     "Spain"  "stays"  "mainly"
[7] "in"     "the"    "plain"
```

Look at the regular expression. The caret says “match anything but” and then `[^0-9A-Za-z]` says “any digit, any lower-case letter, and any upper-case letter.” So the whole expression (including the fact that it is the value of the argument `split`) says “match anything but any digit, any lower-case letter, or any upper-case letter.”

Now we apply this to `moby_dick`.

```
> moby_dick <- unlist(strsplit(moby_dick, split = "[^0-9A-Za-z]"))
```

Let’s see a bit of what we have.

```
> moby_dick[1:50]
[1] "chapter"    "1"          ""           "loomings"
[5] ""           "call"       "me"         "ishmael"
[9] ""           "some"      "years"     "ago"
[13] ""          "never"     "mind"     "how"
```

```
[17] "long"      "precisely"   ""          "having"
[21] "little"    "or"          "no"         "money"
[25] "in"         "my"          "purse"     ""
[29] "and"        "nothing"    "particular" "to"
[33] "interest"   "me"          "on"         "shore"
[37] ""           "i"           "thought"   "i"
[41] "would"     "sail"        "about"     "a"
[45] "little"    "and"         "see"       "the"
[49] "watery"    "part"
```

There is a small problem: Some of the “words” are blank. The following small example indicates why this happened.

```
> unlist(strsplit(c("the rain", "in Spain stays mainly", "in",
+      "the plain"), split = "[^0-9A-Za-z]"))
[1] "the"      "rain"      "in"       "Spain"    ""         ""
[7] ""         "stays"     "mainly"   "in"       "the"     "plain"
```

It is not too hard to remove the blank words.

```
> length(moby_dick)
[1] 253993
> not.blanks <- which(moby_dick != "")
> moby_dick <- moby_dick[not.blanks]
> length(moby_dick)
[1] 214889
> moby_dick[1:50]
[1] "chapter"    "1"          "loomings"   "call"
[5] "me"          "ishmael"    "some"       "years"
[9] "ago"         "never"     "mind"      "how"
[13] "long"        "precisely"  "having"    "little"
[17] "or"          "no"         "money"    "in"
[21] "my"          "purse"     "and"       "nothing"
[25] "particular" "to"        "interest"  "me"
[29] "on"          "shore"     "i"         "thought"
[33] "i"           "would"     "sail"      "about"
[37] "a"           "little"    "and"       "see"
[41] "the"         "watery"   "part"     "of"
[45] "the"         "world"    "it"        "is"
[49] "a"           "way"
```

(In this example it would have been more efficient to replace

```
moby_dick <- unlist(strsplit(moby_dick, split = "[^0-9A-Za-z]"))
```

by

```
moby_dick2 <- unlist(strsplit(moby_dick, split = " ", fixed = TRUE))
```

Then the second step of selecting the non-blank words would not have been necessary. But regular expressions will be essential going forward, so it was worthwhile using regular expressions even if they do not provide the most efficient method.)

## nchar again

Now that the vector `moby_dick` contains each word in the novel as a separate element, it is relatively easy to do some basic analyses. For example the `nchar` function can give us a count of the number of characters in each element of the vector, i.e., can give us the number of letters in each word in the novel.

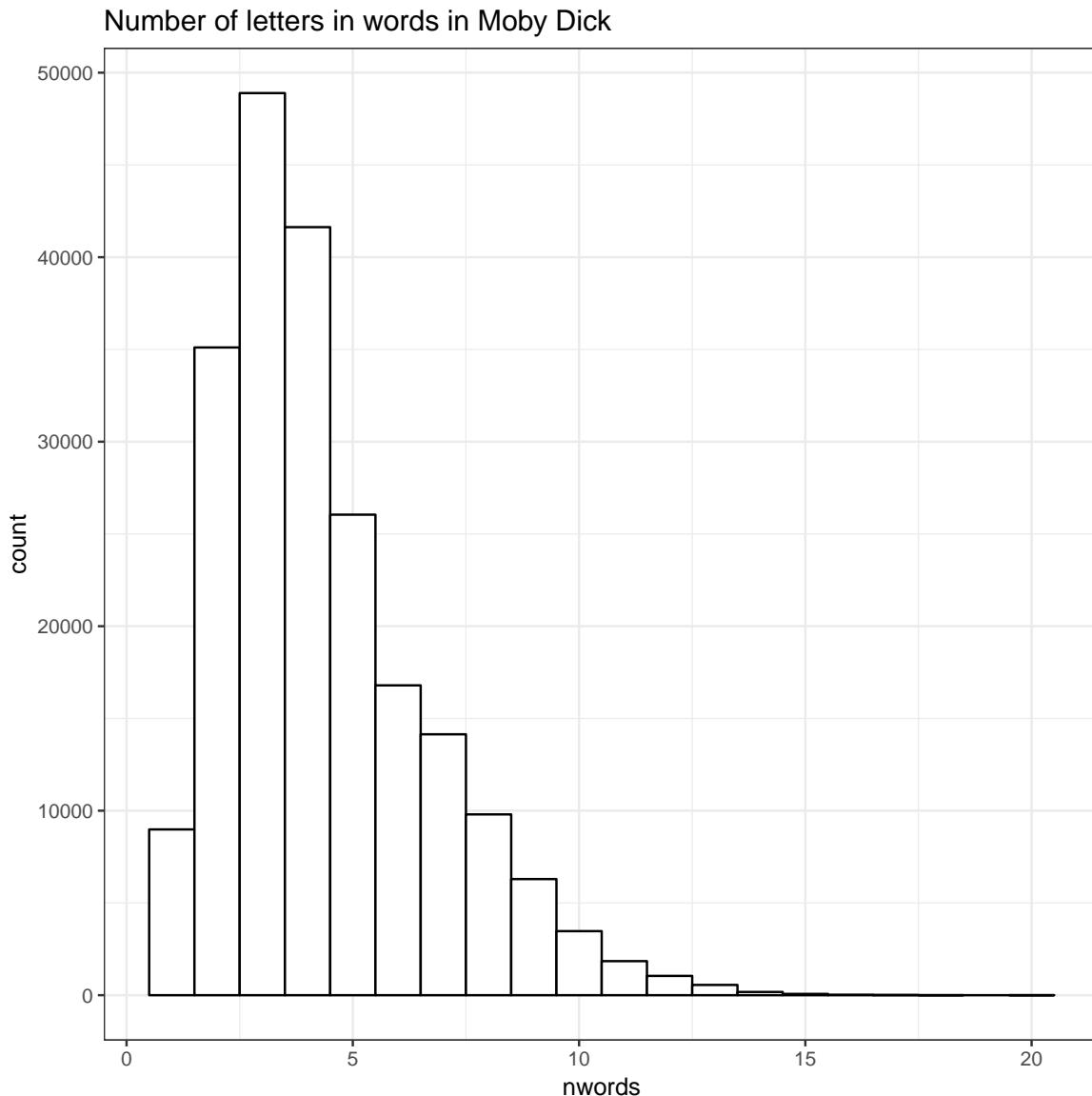
```
> moby_dick_nchar <- nchar(moby_dick)
> moby_dick_nchar[1:50]

[1] 7 1 8 4 2 7 4 5 3 5 4 3 4 9 6 6 2 2
[19] 5 2 2 5 3 7 10 2 8 2 2 5 1 7 1 5 4 5
[37] 1 6 3 3 3 6 4 2 3 5 2 2 1 3

> max(moby_dick_nchar)

[1] 20

> ggplot(data = data.frame(nwords = moby_dick_nchar), aes(x = nwords)) +
+   geom_histogram(binwidth = 1, color = "black", fill = "white") +
+   ggtitle("Number of letters in words in Moby Dick") + theme_bw()
```



```
> moby_dick_word_table <- table(moby_dick)
> moby_dick_word_table <- sort(moby_dick_word_table, decreasing = TRUE)
> moby_dick_word_table[1:50]

moby_dick
   the    of    and     a    to    in    that    it    his     i
14175  6469  6325  4636  4539  4077  3045  2497  2495  2114
      he    but      s     as  with     is    was    for    all  this
1876   1805  1737  1720  1692  1690  1627  1593  1515  1382
      at    by whale   not  from   him    so    on    be    one
1304   1175  1150  1142  1072  1058  1053  1040  1032  907
      you there    now   had  have    or  were  they  like   me
```

|      |       |      |      |       |      |     |     |     |     |
|------|-------|------|------|-------|------|-----|-----|-----|-----|
| 884  | 854   | 779  | 767  | 754   | 689  | 677 | 649 | 639 | 630 |
| then | which | what | some | their | when | an  | are | my  | no  |
| 628  | 625   | 611  | 608  | 604   | 601  | 590 | 587 | 587 | 581 |

### substr and substrim

The **substr** function can be used to extract or replace substrings. The first argument is the string to be manipulated, and the second and third arguments specify the first and last elements of the string to be extracted or to be replaced.

```
> x <- "Michigan"
> substr(x, 3, 4)

[1] "ch"

> substr(x, 3, 4) <- "CH"
> x

[1] "MiCHigan"

> x <- c("Ohio", "Michigan", "Illinois", "Wisconsin")
> substr(x, 2, 4)

[1] "hio" "ich" "lli" "isc"

> substr(x, 2, 4) <- "$#&"
> x

[1] "O$#&"      "M$#&igan"   "I$#&nois"   "W$#&onsin"
```

The **strtrim** function trims a character string to a specified length.

```
> strtrim("Michigan", 1)

[1] "M"

> strtrim("Michigan", 4)

[1] "Mich"

> strtrim("Michigan", 100)
```

```
[1] "Michigan"

> strtrim(c("Ohio", "Michigan", "Illinois", "Wisconsin"), 3)
[1] "Ohi" "Mic" "Ill" "Wis"

> strtrim(c("Ohio", "Michigan", "Illinois", "Wisconsin"), c(3,
+     4, 5, 6))
[1] "Ohi"      "Mich"     "Illin"    "Wiscon"
```

## grep and related functions

The **grep** function searches for a specified pattern and returns either the locations where this pattern is found or the selected elements. The **grepl** function returns a logical vector rather than locations of elements.

Here are some examples. All use **fixed = TRUE** since at this point we are using fixed character strings rather than regular expressions.

```
> grep("a", c("the rain", "in Spain      stays mainly", "in", "the plain"),
+       fixed = TRUE)
[1] 1 2 4

> grep("a", c("the rain", "in Spain      stays mainly", "in", "the plain"),
+       fixed = TRUE, value = TRUE)
[1] "the rain"           "in Spain      stays mainly"
[3] "the plain"

> grepl("a", c("the rain", "in Spain      stays mainly", "in", "the plain"),
+       fixed = TRUE)
[1] TRUE  TRUE FALSE  TRUE
```

The **sub** and **gsub** functions replace a specified character string. The **sub** function only replaces the first occurrence, while **gsub** replaces all occurrences.

```
> gsub("a", "?", c("the rain", "in Spain      stays mainly", "in",
+       "the plain"), fixed = TRUE)
```

```
[1] "the r?in"           "in Sp?in     st?ys m?inly"
[3] "in"                 "the pl?in"

> sub("a", "?", c("the rain", "in Spain     stays mainly", "in",
+      "the plain"), fixed = TRUE)

[1] "the r?in"           "in Sp?in     stays mainly"
[3] "in"                 "the pl?in"

> gsub("a", "???", c("the rain", "in Spain     stays mainly", "in",
+      "the plain"), fixed = TRUE)

[1] "the r???in"
[2] "in Sp???in     st???ys m???inly"
[3] "in"
[4] "the pl???in"

> sub("a", "???", c("the rain", "in Spain     stays mainly", "in",
+      "the plain"), fixed = TRUE)

[1] "the r???in"
[2] "in Sp???in     stays mainly"
[3] "in"
[4] "the pl???in"
```

### Exercise 13

Learning objectives: read and write text data; concatenate text with the `paste` function; analyze text with `nchar`; practice with functions; manipulate strings with `substr` and `strtrim`.

# Chapter 11

## Spatial data visualization and analysis

### 11.1 Overview

Recall, a data structure is a format for organizing and storing data. The structure is designed so that data can be accessed and worked with in specific ways. Statistical software and programming languages have methods (or functions) designed to operate on different kinds of data structures.

This chapter focuses on spatial data structures and some of the R functions that work with these data. Spatial data comprise values associated with locations, e.g., temperature at a given latitude, longitude, and perhaps elevation. Spatial data are typically organized into **vector** or **raster** data types. (See Figure 11.1.)

- Vector data represent features such as discrete points, lines, and polygons.
- Raster data represent surfaces as a rectangular matrix of square cells or pixels.

Depending on the type of problem that needs to be solved, the type of maps that need to be made, and the data source, either vector or raster, or a combination of the two, is used. Each data structure has strengths and weaknesses in terms of functionality and representation. As you gain more experience working with spatial data, you will be able to determine which structure to use for a particular application.

There is a large set of R packages available for working with spatial (and space-time) data. These packages are described in the [CRAN Task View: Analysis of Spatial Data](#). The CRAN task view attempts to organize the various packages into categories, e.g., *Handling spatial data*, *Reading and writing spatial data*, *Visualization*, and *Disease mapping and areal data analysis*, so users can quickly identify package options given their project needs.

Exploring the extent of the excellent spatial data tools available in R is beyond the scope of this book. Rather, we would point you to subject texts like *Applied*

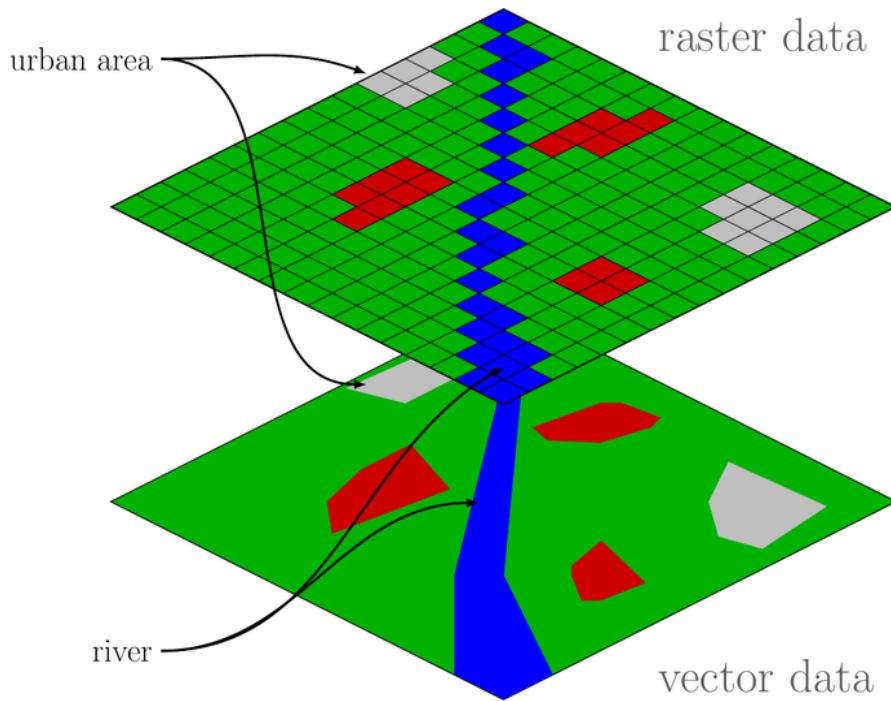


Figure 11.1: Raster/vector comparison ([Wegmann \[2010\]](#))

*Spatial Data Analysis with R* by [Bivand et al. \[2013\]](#) available for free via the MSU library system, and numerous online tutorials on pretty much any aspect of spatial data analysis with R. These tools make R a full-blown [Geographic Information System \(GIS\)](#) capable of spatial data manipulation and analysis on par with commercial GIS systems such as [ESRI's ArcGIS](#).

### 11.1.1 Some spatial data packages

This chapter will focus on a few R packages for manipulating and visualizing spatial data. Specifically we will touch on the following packages

- **sp**: spatial data structures and methods
- **rgdal**: interface to the C/C++ spatial data Geospatial Data Abstraction Library
- **ggmap**: extends **ggplot2** language to handle spatial data
- **leaflet**: generates dynamic online maps

## 11.2 Motivating data

We motivate the topics introduced in this chapter using some forestry data from the [Penobscot Experimental Forest](#) (PEF) located in Maine. The PEF is a long-term experimental forest that is used to understand the effects of silviculture (i.e., science of tending trees) treatments on forest growth and composition. The PEF is divided into non-overlapping management units that receive different harvesting treatments. Within each management unit is a series of observation point locations (called forest inventory plots) where forest variables have been measured. Ultimately, we want to summarize the inventory plots measurements by management unit and map the results.

## 11.3 Reading spatial data into R

Spatial data come in a variety of file formats. Examples of popular vector file formats for points, lines, and polygons, include ESRI's [shapefile](#) and open standard [GeoJSON](#). Common raster file formats include [GeoTIFF](#) and [netCDF](#)<sup>1</sup>.

The `rgdal` function `readOGR` will read a large variety of vector data file formats (there is also a `writeOGR()` for writing vector data files). Raster data file formats can be read using the `rgdal` function `readGDAL` (yup, also a `writeGDAL()`) or read functions in the `raster` package. All of these functions automatically cast the data into an appropriate R spatial data object (i.e., data structure), which are defined in the `sp` or `raster` packages. Table 11.1 provides an abbreviated list of these R spatial objects<sup>2</sup>. The *Without attributes* column gives the `sp` package's spatial data object classes for points, lines, polygons, and raster pixels that do not have data associated with the spatial objects (i.e., without attributes in GIS speak). `DataFrame` is appended to the object class name once data, in the form of variables, are added to the spatial object.

You can create your own spatial data objects in R. Below, for example, we create a `SpatialPoints` object consisting of four points. Then add some data to the points to make it a `SpatialPointsDataFrame`.

```
> library(sp)
>
> x <- c(3, 2, 5, 6)
> y <- c(2, 5, 6, 7)
>
> coords <- cbind(x, y)
>
```

---

<sup>1</sup>A longer list of spatial data file formats is available at [https://en.wikipedia.org/wiki/GIS\\_file\\_formats](https://en.wikipedia.org/wiki/GIS_file_formats).

<sup>2</sup>A more complete list of the `sp` package's spatial data classes and methods is detailed in the package's vignette [https://cran.r-project.org/web/packages/sp/vignettes/intro\\_sp.pdf](https://cran.r-project.org/web/packages/sp/vignettes/intro_sp.pdf).

|          | Without attributes | With attributes          |
|----------|--------------------|--------------------------|
| Polygons | SpatialPolygons    | SpatialPolygonsDataFrame |
| Points   | SpatialPoints      | SpatialPointsDataFrame   |
| Lines    | SpatialLines       | SpatialLinesDataFrame    |
| Raster   | SpatialGrid        | SpatialGridDataFrame     |
| Raster   | SpatialPixels      | SpatialPixelsDataFrame   |
| Raster   |                    | RasterLayer              |
| Raster   |                    | RasterBrick              |
| Raster   |                    | RasterStack              |

Table 11.1: An abbreviated list of `sp` and `raster` data objects and associated classes for the fundamental spatial data types.

```
> sp.pts <- SpatialPoints(coords)
>
> class(sp.pts)

[1] "SpatialPoints"
attr(,"package")
[1] "sp"

> some.data <- data.frame(var.1 = c("a", "b", "c", "d"), var.2 = 1:4)
>
> sp.pts.df <- SpatialPointsDataFrame(sp.pts, some.data)
>
> class(sp.pts.df)

[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

If, for example, you already have a data frame that includes the spatial coordinate columns and other variables, then you can promote it to a `SpatialPointsDataFrame` by indicating which columns contain point coordinates. You can extract or access the data frame associated with the spatial object using `@data`. You can also access individual variables directly from the spatial object using `$` or by name or column number to the right of the comma in `[,]` (analogues to accessing variables in a data frame).

```
> df <- data.frame(x = c(3, 2, 5, 6), y = c(2, 5, 6, 7), var.1 = c("a",
+      "b", "c", "d"), var.2 = 1:4)
> class(df)

[1] "data.frame"
```

```
> # promote to a SpatialPointsDataFrame
> coordinates(df) <- ~x + y
>
> class(df)

[1] "SpatialPointsDataFrame"
attr("package")
[1] "sp"

> # access entire data frame
> df@data

  var.1 var.2
1     a     1
2     b     2
3     c     3
4     d     4

> class(df@data)

[1] "data.frame"

> # access columns directly
> df$var.1

[1] a b c d
Levels: a b c d

> df[, c("var.1", "var.2")]

class      : SpatialPointsDataFrame
features   : 4
extent     : 2, 6, 2, 7  (xmin, xmax, ymin, ymax)
coord. ref. : NA
variables  : 2
names      : var.1, var.2
min values :     a,     1
max values :     d,     4

> df[, 2]
```

```

class      : SpatialPointsDataFrame
features    : 4
extent      : 2, 6, 2, 7  (xmin, xmax, ymin, ymax)
coord. ref. : NA
variables   : 1
names       : var.2
min values  :      1
max values  :      4

> # get the bounding box
> bbox(df)

      min max
x    2    6
y    2    7

```

Here, the data frame `df` is promoted to a `SpatialPointsDataFrame` by indicating the column names that hold the longitude and latitude (i.e., `x` and `y` respectively) using the `coordinates` function. Here too, the `@data` is used to retrieve the data frame associated with the points. We also illustrate how variables can be accessed directly from the spatial object. The `bbox` function is used to return the bounding box that is defined by the spatial extent of the point coordinates. The other spatial objects noted in Table 11.1 can be created, and their data accessed, in a similar way<sup>3</sup>.

More than often we find ourselves reading existing spatial data files into R. The code below uses the `downloader` package to download all of the PEF data we'll use in this chapter. The data are compressed in a single zip file, which is then extracted into the working directory using the `unzip` function. A look into the PEF directory using `list.files` shows nine files<sup>4</sup>. Those named “MU-bounds.\*” comprise the shapefile that holds the PEF’s management unit boundaries in the form of polygons. Like other spatial data file formats, shapefiles are made up of several different files (with different file extensions) that are linked together to form a spatial data object. The “plots.csv” file holds the spatial coordinates and other information about the PEF’s forest inventory plots. The “roads.\*” shapefile holds roads and trails in and around the PEF.

```

> library(downloader)
>
> download("http://blue.for.msu.edu/FOR875/data/PEF.zip",
+           destfile=".~/PEF.zip", mode="wb")
>

```

---

<sup>3</sup>This cheatsheet written by Barry Rowlingson is a useful reference [www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html](http://www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html)

<sup>4</sup>The `list.files` function does not read data into R; it simply prints the contents of a directory.

```
> unzip("PEF.zip", exdir = ".")
>
> list.files("PEF")

[1] "MU-bounds.dbf" "MU-bounds.prj" "MU-bounds.qpj"
[4] "MU-bounds.shp" "MU-bounds.shx" "plots.csv"
[7] "plots.dbf"      "plots.prj"     "plots.qpj"
[10] "plots.shp"     "plots.shx"     "roads.dbf"
[13] "roads.prj"     "roads.shp"    "roads.shx"
```

Next we read the MU-bounds shapefile into R using `readOGR()`<sup>5</sup> and explore the resulting `mu` object. Notice that when we read a shapefile into R, we do not include a file extension with the shapefile name because a shapefile is always composed of multiple files.

```
> library(rgdal)
> mu <- readOGR("PEF", "MU-bounds")

OGR data source with driver: ESRI Shapefile
Source: "C:\Users\Shawn Santo\Desktop\for875-master\for875-master\sp\PEF", layer: "MU-
with 40 features
It has 1 fields
```

When called, the `readOGR` function provides a bit of information about the object being read in. Here, we see that it read the MU-bounds shapefile from PEF directory and the shapefile had 40 features (i.e., polygons) and 1 field (i.e., field is synonymous with column or variable in the data frame).

You can think of the resulting `mu` object as a data frame where each row corresponds to a polygon and each column holds information about the polygon<sup>6</sup>. More specifically, the `mu` object is a `SpatialPolygonsDataFrame`.

```
> class(mu)

[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

As illustrated using the made-up point data in the example above, you can access the data frame associated with the polygons using `@data`.

---

<sup>5</sup>The authors of the `rgdal` library decided to have some information about the version of GDAL and other software specifics be printed when the library is loaded. Don't let it distract you.

<sup>6</sup>Much of the actual spatial information is hidden from you in other parts of the data structure, but is available if you ask nicely for it (see subsequent sections).

```
> class(mu@data)
[1] "data.frame"

> dim(mu@data)
[1] 40 1

> head(mu@data)

  mu_id
0   C15
1   C17
2   C16
3   C27
4   U18
5   U31
```

Above, a call to **class()** confirms we have accessed the data frame, **dim()** shows there 40 rows (one row for each polygon) and one column, and **head()** shows the first six values of the column named **mu\_id**. The **mu\_id** values are unique identifiers for each management unit polygon across the PEF.

## 11.4 Coordinate reference systems

One of the more challenging aspects of working with spatial data is getting used to the idea of a coordinate reference system. A *coordinate reference system* (CRS) is a system that uses one or more numbers, or coordinates, to uniquely determine the position of a point or other geometric element (e.g., line, polygon, raster). For spatial data, there are two common coordinate systems:

1. Spherical coordinate system such as latitude-longitude, often referred to as a *geographic coordinate system*.
2. Projected coordinate system based on a map projection which is a systematic transformation of the latitudes and longitudes that aims to minimize distortion that occurs from projecting maps of the earth's spherical surface onto a two-dimensional Cartesian coordinate plane. Projected coordinate systems are sometimes referred to as *map projections*.

There are numerous map projections<sup>7</sup>. One of the more frustrating parts of working with spatial data is that it seems like each data source you find offers its data in a

---

<sup>7</sup>See partial list of map projections at [https://en.wikipedia.org/wiki/List\\_of\\_map\\_projections](https://en.wikipedia.org/wiki/List_of_map_projections). See a humorous discussion of map projections at <http://brilliantmaps.com/xkcd/>.

different map projection and hence you spend a great deal of time *reprojecting* (i.e., transforming from one CRS to another) data into a common CRS such that they overlay correctly. Reprojecting is accomplished using the `sp` package's `spTransform` function as demonstrated in Section 11.5.

In R, a spatial object's CRS is accessed via the `sp` package `proj4string` function. The code below shows the current projection of `mu`.

```
> proj4string(mu)
[1] "+proj=utm +zone=19 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0"
```

The cryptic looking string returned by `proj4string()` is a set of directives understood by the `proj.4` C library, which is part of `sp`, and used to map geographic longitude and latitude coordinates into the projected Cartesian coordinates. This CRS tells us the `mu` object is in [Universal Transverse Mercator \(UTM\)](#) zone 19 coordinate system.<sup>8</sup>

## 11.5 Illustration using `ggmap`

Let's begin by making a map of PEF management unit boundaries over top of a satellite image using the `ggmap` package. Given an address, location, or bounding box, the `ggmap` package's `get_map` function will query Google Maps, OpenStreetMap, Stamen Maps, or Naver Map servers for a user-specified map type. The `get_map` function requires the location or bounding box coordinates be in a geographic coordinate system (i.e., latitude-longitude). This means we need to reproject `mu` from UTM zone 19 to latitude-longitude geographic coordinates, which is defined by the "`+proj=longlat +datum=WGS84`" `proj.4` string. As seen below, the first argument in `spTransform` function is the spatial object to reproject and the second argument is a CRS object created by passing a `proj.4` string into the `CRS` function.

```
> mu <- spTransform(mu, CRS("+proj=longlat +datum=WGS84"))
> proj4string(mu)
[1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

Unfortunately, we cannot just feed the `SpatialPolygonsDataFrame` `mu` into `ggplot` (perhaps some day soon this will be possible). Rather, we need to first convert the `SpatialPolygonsDataFrame` into a specially formatted data frame using the `fortify` function that is part of the `ggplot2` package<sup>9</sup>. The `fortify` function will also need a

---

<sup>8</sup>If you start dealing with a lot of spatial data and reprojecting, <http://spatialreference.org> is an excellent resource for finding and specifying coordinate reference systems.

<sup>9</sup>`ggmap` depends on `ggplot2` so `ggplot2` will be automatically loaded when you call `library(ggmap)`.

unique identifier for each polygon specified using the `region` argument, which for `mu` is the `mu_id`.

```
> library(ggmap)
> mu.f <- fortify(mu, region = "mu_id")
> head(mu.f)

  long      lat order hole piece id group
1 -68.62065 44.85504     1 FALSE    1 C12 C12.1
2 -68.61809 44.85817     2 FALSE    1 C12 C12.1
3 -68.61810 44.85813     3 FALSE    1 C12 C12.1
4 -68.61812 44.85802     4 FALSE    1 C12 C12.1
5 -68.61811 44.85797     5 FALSE    1 C12 C12.1
6 -68.61812 44.85791     6 FALSE    1 C12 C12.1
```

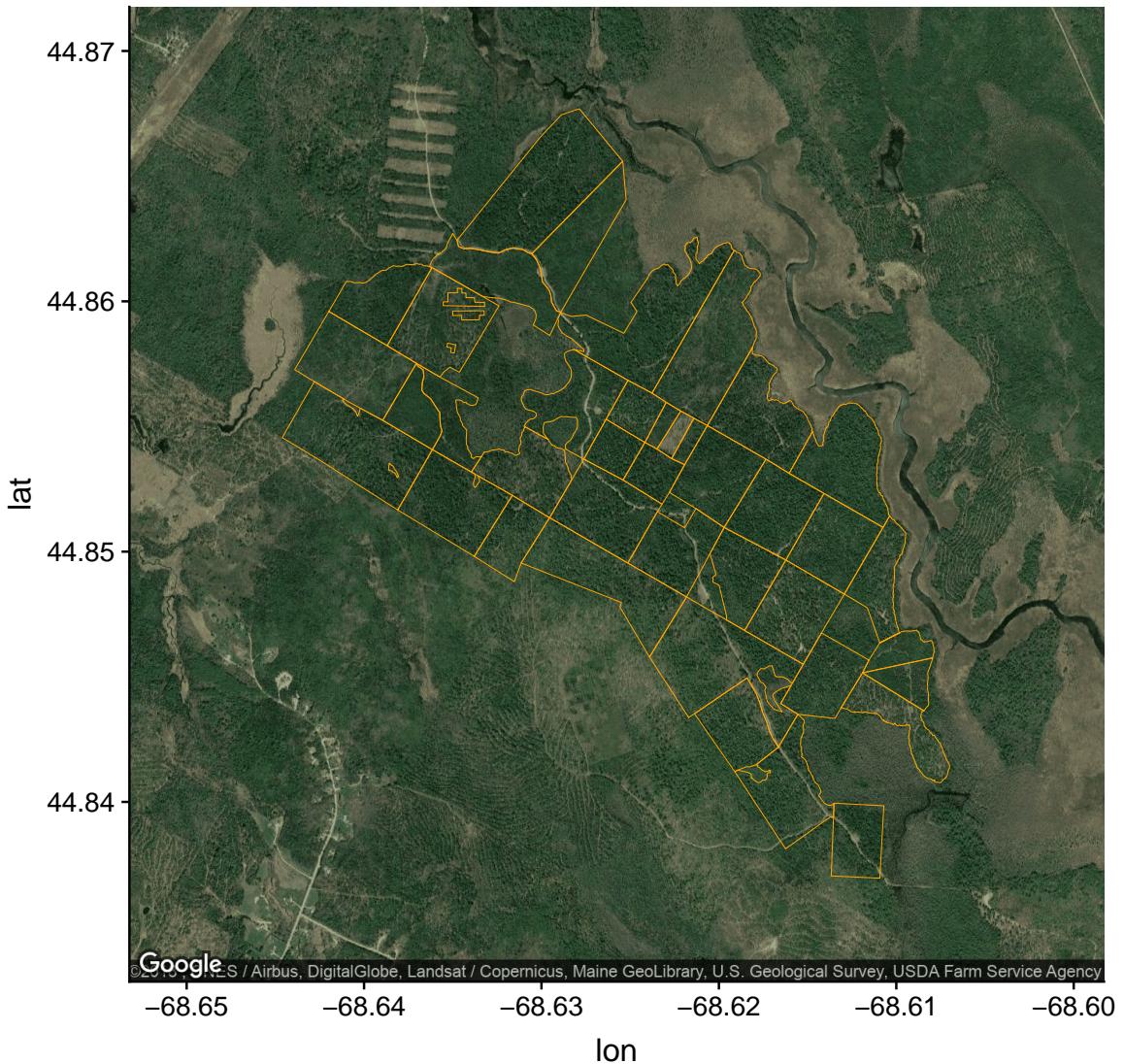
Notice the `id` column in the fortified version of `mu` holds each polygon's `mu_id` value (this will be important later when we link data to the polygons).

Next, we query the satellite imagery used to underlay the management units (we'll generally refer to this underlying map as the basemap).

```
> mu.bbox <- bbox(mu)
>
> basemap <- get_map(location=mu.bbox, zoom = 14, maptype="satellite")

Warning: bounding box given to google - spatial extent only approximate.
converting bounding box to center/zoom specification. (experimental)
Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=44.852323,-68.62065&z=14&t=s

> ggmap(basemap) +
+   geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+               fill=NA, size=0.2, color="orange")
```



Looks pretty good!<sup>10</sup> Take a look at the `get_map` function manual page and try different options for `maptype` (e.g., `maptype="terrain"`).

Next we'll add the forest inventory plots to the map. Begin by reading in the PEF forest inventory plot data held in "plots.csv". Recall, foresters have measured forest variables at a set of locations (i.e., inventory plots) within each management unit. The following statement reads these data and displays the resulting data frame structure.

```
> plots <- read.csv("PEF/plots.csv", stringsAsFactors = FALSE)
> str(plots)

'data.frame': 451 obs. of  8 variables:
```

<sup>10</sup>You can ignore the warning and messages.

```
$ mu_id          : chr  "U10" "U10" "U10" "U10" ...
$ plot           : int   11 13 21 22 23 24 31 32 33 34 ...
$ easting        : num   529699 529777 529774 529814 529850 ...
$ northing       : num   4966333 4966471 4966265 4966336 4966402 ...
$ biomass_mg_ha  : num   96.3 115.7 121.6 72 122.3 ...
$ stems_ha        : int   5453 2629 3385 7742 7980 10047 5039 5831 2505 7325 ...
$ diameter_cm     : num   4.8 6.9 6.1 3.1 4.7 1.6 4.1 5.2 5.7 3.3 ...
$ basal_area_m2_ha: num   22 23.2 23 16.1 29.2 19.1 14.1 27.4 21.6 15 ...
```

In `plots` each row is a forest inventory plot and columns are:

1. `mu_id` identifies the management unit within which the plot is located
2. `plot` unique plot number within the management unit
3. `easting` longitude coordinate in UTM zone 19
4. `northing` latitude coordinate in UTM zone 19
5. `biomass_mg_ha` tree biomass in metric ton (per hectare basis)
6. `stocking_stems_ha` number of tree (per hectare basis)
7. `diameter_cm` average tree diameter measured 130 cm up the tree trunk
8. `basal_area_m2_ha` total cross-sectional area at 130 cm up the tree trunk (per hectare basis)

There is nothing inherently spatial about this data structure—it is simply a data frame. We make `plots` into a spatial object by identifying which columns hold the coordinates. This is done below using the `coordinates` function, which promotes the `plots` data frame to a `SpatialPointsDataFrame`.

```
> coordinates(plots) <- ~easting + northing
>
> class(plots)

[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

Although `plots` is now a `SpatialPointsDataFrame`, it does not know to which CRS the coordinates belong; hence, the `NA` when `proj4string(plots)` is called below. As noted in the `plots` file description above, `easting` and `northing` are in UTM zone 19. This CRS is set using the second call to `proj4string` below.

```
> proj4string(plots)

[1] NA

> proj4string(plots) <- CRS("+proj=utm +zone=19 +datum=NAD83 +units=m
+no_defs +ellps=GRS80 +towgs84=0,0,0")
```

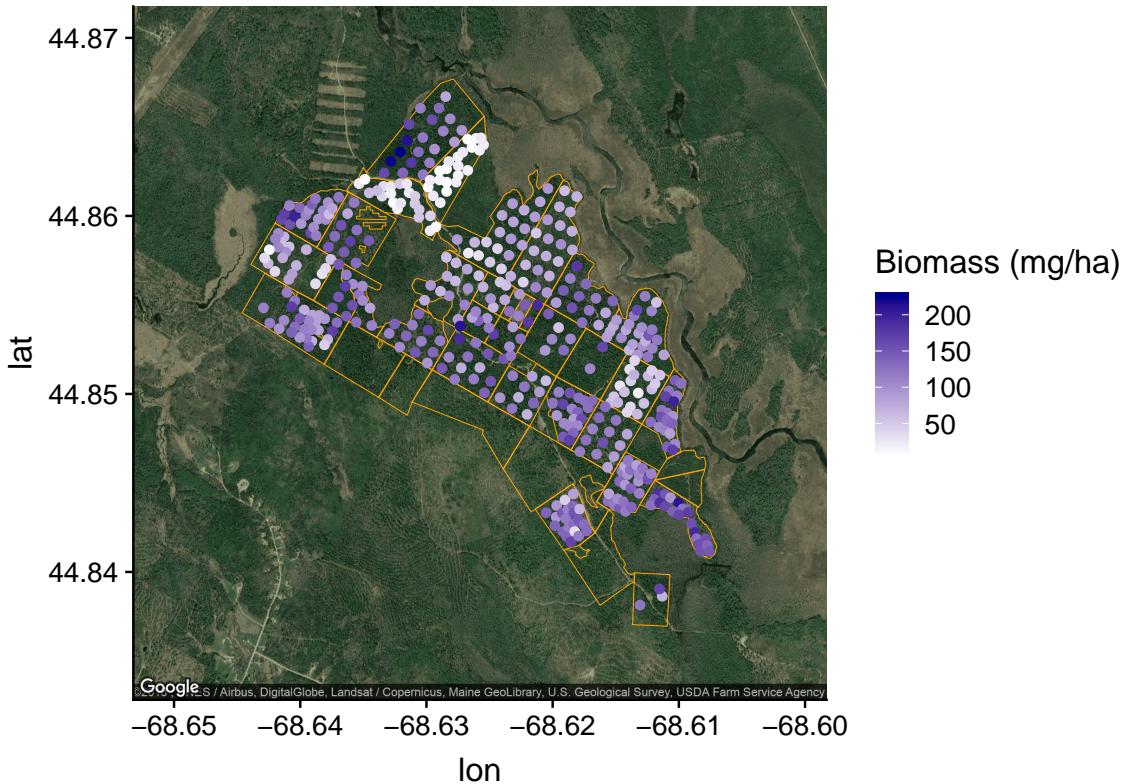
Now let's reproject `plots` to share a common CRS with `mu`

```
> plots <- spTransform(plots, CRS("+proj=longlat +datum=WGS84"))
```

Note, because `mu` is already in the projection we want for `plots`, we could have replaced second argument in the `spTransform` call above with `proj4string(mu)` and saved some typing.

We're now ready to add the forest inventory plots to the existing basemap with management units. Specifically, let's map the `biomass_mg_ha` variable to show changes in biomass across the forest. No need to fortify `plots`, `ggplot` is happy to take `geom_point`'s `data` argument as a data frame (although we do need to convert `plots` from a `SpatialPointsDataFrame` to a data frame using the `as.data.frame` function). Check out the `scale_color_gradient` function in your favorite `ggplot2` reference to understand how the color scale is set.

```
> ggmap(basemap) +
+   geom_polygon(data=mu.f, aes(x = long, y = lat, group=group),
+                 fill=NA, size=0.2, color="orange") +
+   geom_point(data=as.data.frame(plots),
+              aes(x = easting, y = northing, color=biomass_mg_ha)) +
+   scale_color_gradient(low="white", high="darkblue") +
+   labs(color = "Biomass (mg/ha)")
```



There is something subtle and easy to miss in the code above. Notice the `aes` function arguments in `geom_points` take geographic longitude and latitude, `x` and `y` respectively, from the `points` data frame (but recall `easting` and `northing` were in UTM zone 19). This works because we applied `spTransform` to reproject the `points` `SpatialPointsDataFrame` to geographic coordinates. `sp` then replaces the values in `easting` and `northing` columns with the reprojected coordinate values when converting a `SpatialPointsDataFrame` to a data frame via `as.data.frame()`.

Foresters use the inventory plot measurements to estimate forest variables within management units, e.g., the average or total management unit biomass. Next we'll make a plot with management unit polygons colored by average `biomass_mg_ha`.

```
> mu.bio <- plots@data %>% group_by(mu_id) %>%
+   summarize(biomass_mu = mean(biomass_mg_ha))
```

```
> print(mu.bio)

# A tibble: 33 x 2
  mu_id biomass_mu
  <chr>     <dbl>
1 C12      124.
2 C15      49.9
3 C16      128.
4 C17      112.
5 C20      121.
6 C21      134.
7 C22      65.2
8 C23A     108.
9 C23B     153.
10 C24     126.
# ... with 23 more rows
```

Recall from Section 6.5 this one-liner can be read as “get the data frame from `plots`’s `SpatialPointsDataFrame` *then* group by management unit *then* make a new variable called `biomass_mu` that is the average of `biomass_mg_ha` and assign it to the `mu.bio` tibble.”

The management unit specific `biomass_mu` can now be joined to the `mu` polygons using the common `mu_id` value. Remember when we created the fortified version of `mu` called `mu.f`? The `fortify` function `region` argument was `mu_id` which is the `id` variable in the resulting `mu.f`. This `id` variable in `mu.f` can be linked to the `mu_id` variable in `mu.bio` using `dplyr`’s `left_join` function as illustrated below.

```
> head(mu.f, n = 2)

  long      lat order hole piece id group
1 -68.62065 44.85504     1 FALSE    1 C12 C12.1
2 -68.61809 44.85817     2 FALSE    1 C12 C12.1

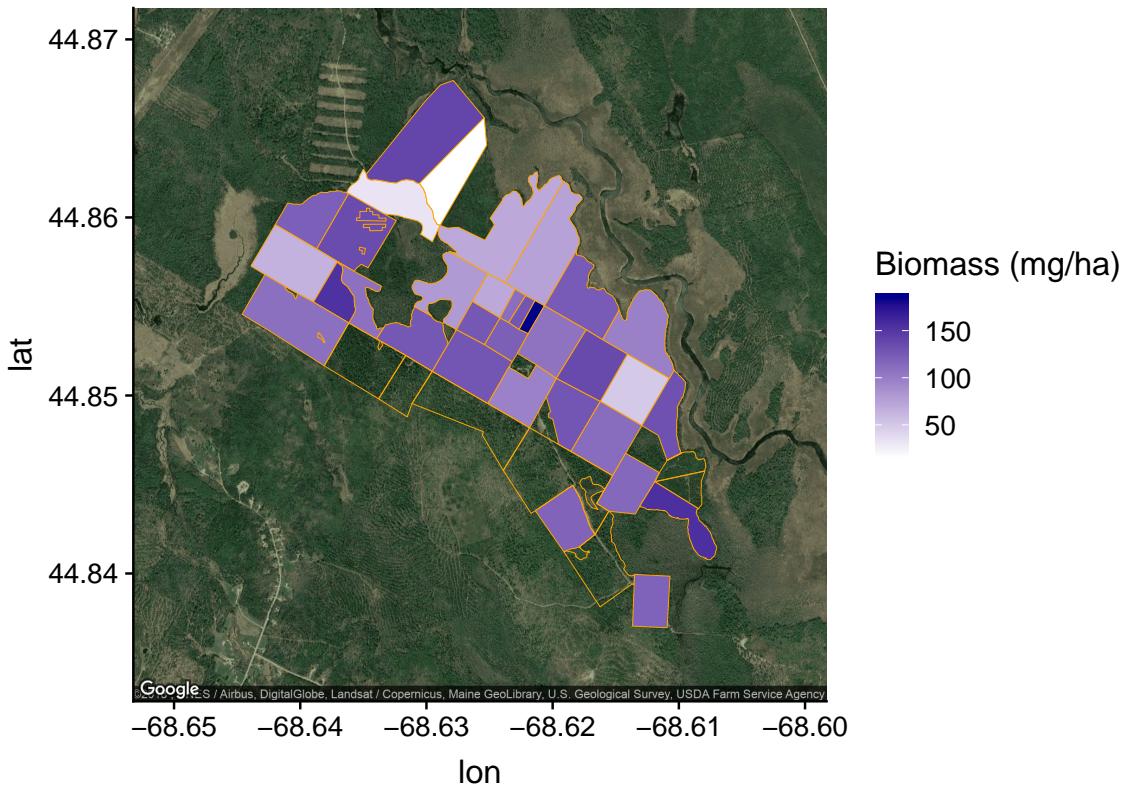
> mu.f <- left_join(mu.f, mu.bio, by = c(id = "mu_id"))

> head(mu.f, n = 2)

  long      lat order hole piece id group biomass_mu
1 -68.62065 44.85504     1 FALSE    1 C12 C12.1 123.7338
2 -68.61809 44.85817     2 FALSE    1 C12 C12.1 123.7338
```

The calls to `head()` show the first few rows of `mu.f` pre- and post-join. After the join, `mu.f` includes `biomass_mu`, which is used below for `geom_polygon`’s `fill` argument to color the polygons accordingly.

```
> ggmap(basemap) +
+   geom_polygon(data=mu.f, aes(x = long, y = lat,
+                               group=group, fill=biomass_mu),
+               size=0.2, color="orange") +
+   scale_fill_gradient(low="white", high="darkblue",
+                       na.value="transparent") +
+   labs(fill="Biomass (mg/ha)")
```



Let's add the roads and some more descriptive labels as a finishing touch. The roads data include a variable called `type` that identifies the road type. To color roads by type in the map, we need to join the `roads` data frame with the fortified roads `roads.f` using the common variable `id` as a road segment specific identifier. Then `geom_path`'s `color` argument gets this `type` variable as a factor to create road-specific

color. The default coloring of the roads blends in too much with the polygon colors, so we manually set the road colors using the `scale_color_brewer` function. The `palette` argument in this function accepts a set of key words, e.g., "Dark2", that specify sets of diverging colors chosen to make map object difference optimally distinct (see, the manual page for `scale_color_brewer`, <http://colorbrewer2.org>, and blog [here](#)).<sup>11</sup>.

```
> roads <- readOGR("PEF", "roads")

OGR data source with driver: ESRI Shapefile
Source: "C:\Users\Shawn Santo\Desktop\for875-master\for875-master\sp\PEF", layer: "roa
with 33 features
It has 2 fields

> roads <- spTransform(roads, proj4string(mu))
>
> roads.f <- fortify(roads, region="id")
> roads.f <- left_join(roads.f, roads@data, by = c('id' = 'id'))

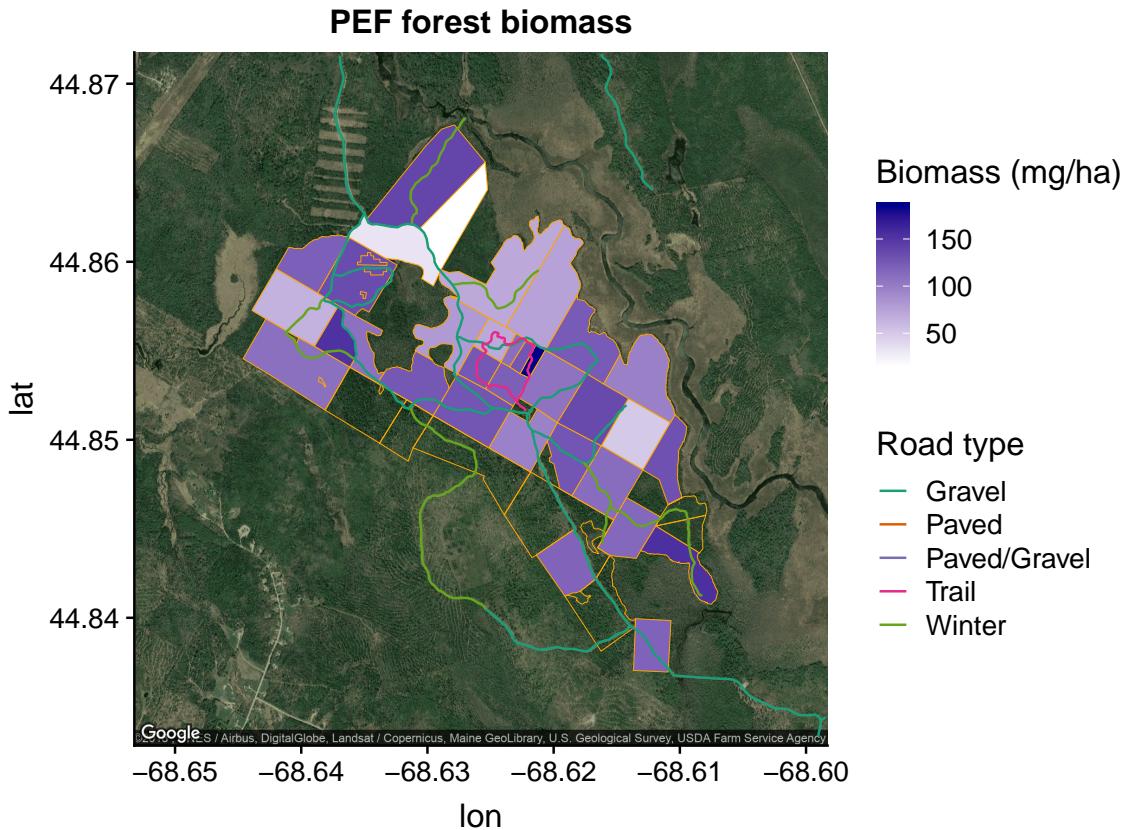
Warning: Column 'id' joining character vector and factor, coercing into
character vector

> ggmap(basemap) +
+   geom_polygon(data=mu.f, aes(x = long, y = lat, group=group,
+                               fill=biomass_mu),
+               size=0.2, color="orange") +
+   geom_path(data=roads.f, aes(x = long, y = lat,
+                             group=group, color=factor(type))) +
+   scale_fill_gradient(low="white", high="darkblue",
+                       na.value="transparent") +
+   scale_color_brewer(palette="Dark2") +
+   labs(fill="Biomass (mg/ha)", color="Road type", xlab="Longitude",
+        ylab="Latitude", title="PEF forest biomass")

Warning: Removed 616 rows containing missing values (geom_path).
```

---

<sup>11</sup>Install the `RColorBrewer` package and run `library(RColorBrewer)`; `display.brewer.all()` to get a graphical list of available palettes.



The second, and more cryptic, of the two warnings from this code occurs because some of the roads extend beyond the range of the map axes and are removed (nothing to worry about).

## 11.6 Illustration using leaflet

Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. As noted on the official [R leaflet website](#), it's used by websites ranging from *The New York Times* and *The Washington Post* to GitHub and Flickr, as well as by GIS specialists like OpenStreetMap, Mapbox, and CartoDB.

The [R leaflet website](#) is an excellent resource to learn leaflet basics, and should serve as a reference to gain a better understanding of the topics we briefly explore

below.

You create a leaflet map using these basic steps:

1. Create a map by calling `leaflet()`
2. Add data layers to the map using layer functions such as, `addTiles()`, `addMarkers()`, `addPolygons()`, `addCircleMarkers()`, `addPolylines()`, `addRasterImage()` and other `add...` functions
3. Repeat step 2 to add more layers to the map
4. Print the map to display it

Here's a brief example.

```
> library(leaflet)
>
> m <- leaflet() %>%
+     addTiles() %>% # Add default OpenStreetMap map tiles
+     addMarkers(lng=-84.482004, lat=42.727516,
+                popup="Here I am!") # Add a clickable marker
> m # Print the map
```

There are a couple things to note in the code. First we use the pipe operator `%>%` just like in `dplyr` functions. Second the `popup` argument in `addMarkers()` takes standard HTML and clicking on the marker makes the text popup. Third the resulting static map in Figure 11.2 does not do justice to how incredibly fun, amazing, and addictive it is to create dynamic web maps (so run the code above in RStudio)!

As seen in the `leaflet()` call above, the various `add...` functions can take longitude (i.e., `lng`) and latitude (i.e., `lat`). Alternatively, these functions can extract the necessary spatial information from `sp` objects, e.g., Table 11.1, when passed to the `data` argument (which greatly simplifies life compared with map making using `ggmap`).

Given that a static PDF document like this does not lend itself to exploring the functionality of leaflet, we will talk more about leaflet's capabilities in the exercise at the end of this chapter.

## 11.7 Subsetting spatial data

You can imagine that we might want to subset spatial objects to map-specific points, lines, or polygons that meet some criteria, or perhaps extract values from polygons or raster surfaces at a set of points or geographic extent. These, and similar types, of operations are easy in R (as long as all spatial objects are in a common CRS). Recall from Section 4.7 how handy it is to subset data structures, e.g., vectors and data frames, using the `[]` operator and logical vectors? Well it's just as easy to subset spatial objects, thanks to the authors of `sp`, `raster`, and other spatial data packages.

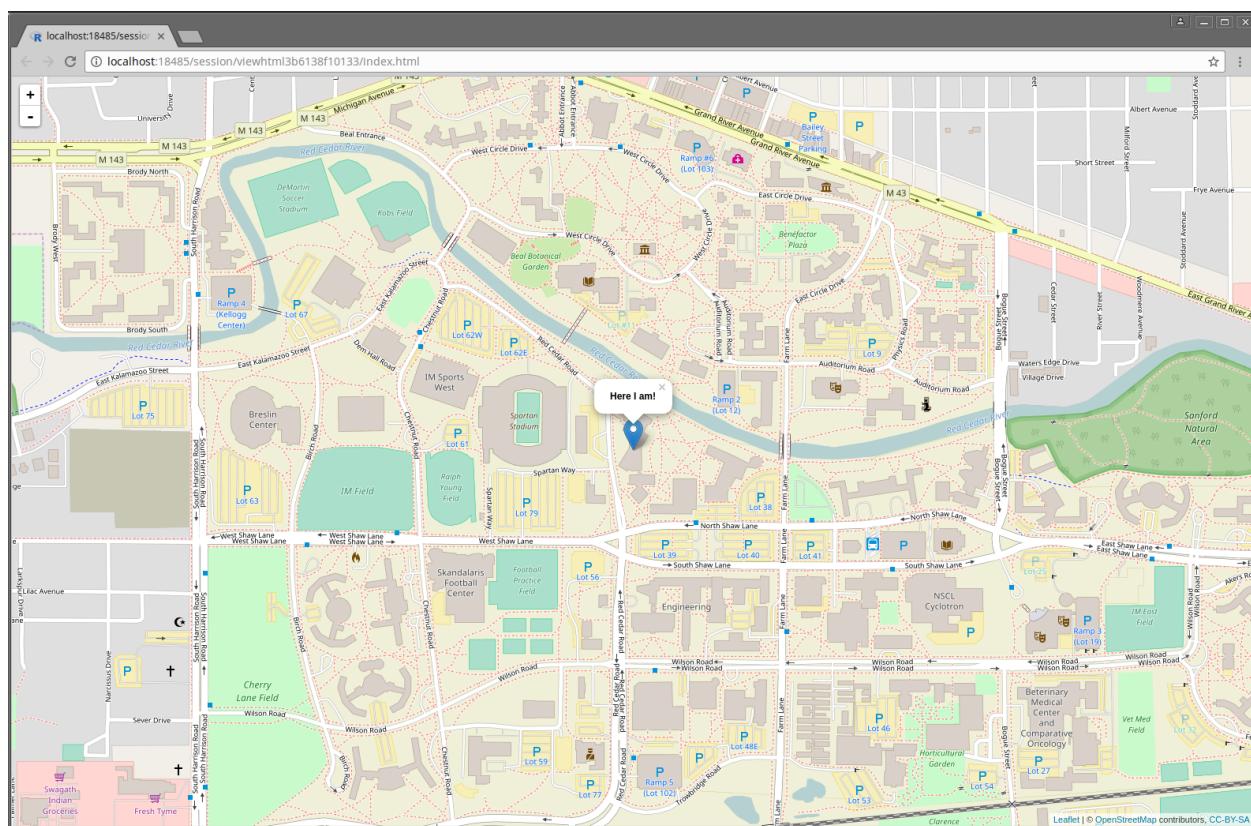


Figure 11.2: leaflet interactive web map.

### 11.7.1 Fetching and cropping data using raster

In order to motivate our exploration of spatial data subsetting and to illustrate some useful functionality of the `raster` package, let's download some elevation data for the PEF. The `raster` package has a rich set of functions for manipulating raster data as well as functions for downloading data from open source repositories. We'll focus on the package's `getData` function, which, given a location in geographic longitude and latitude or location name, will download data from [Global Administrative Boundaries \(GADM\)](#), [Shuttle Radar Topography Mission \(SRTM\)](#), [Global Climate Data \(worldclim\)](#), and other sources commonly in spatial data applications.

Let's download SRTM surface elevation data for the PEF, check the resulting object's class and CRS, and display it using the `raster` package's `image` function along with the PEF forest inventory plots.

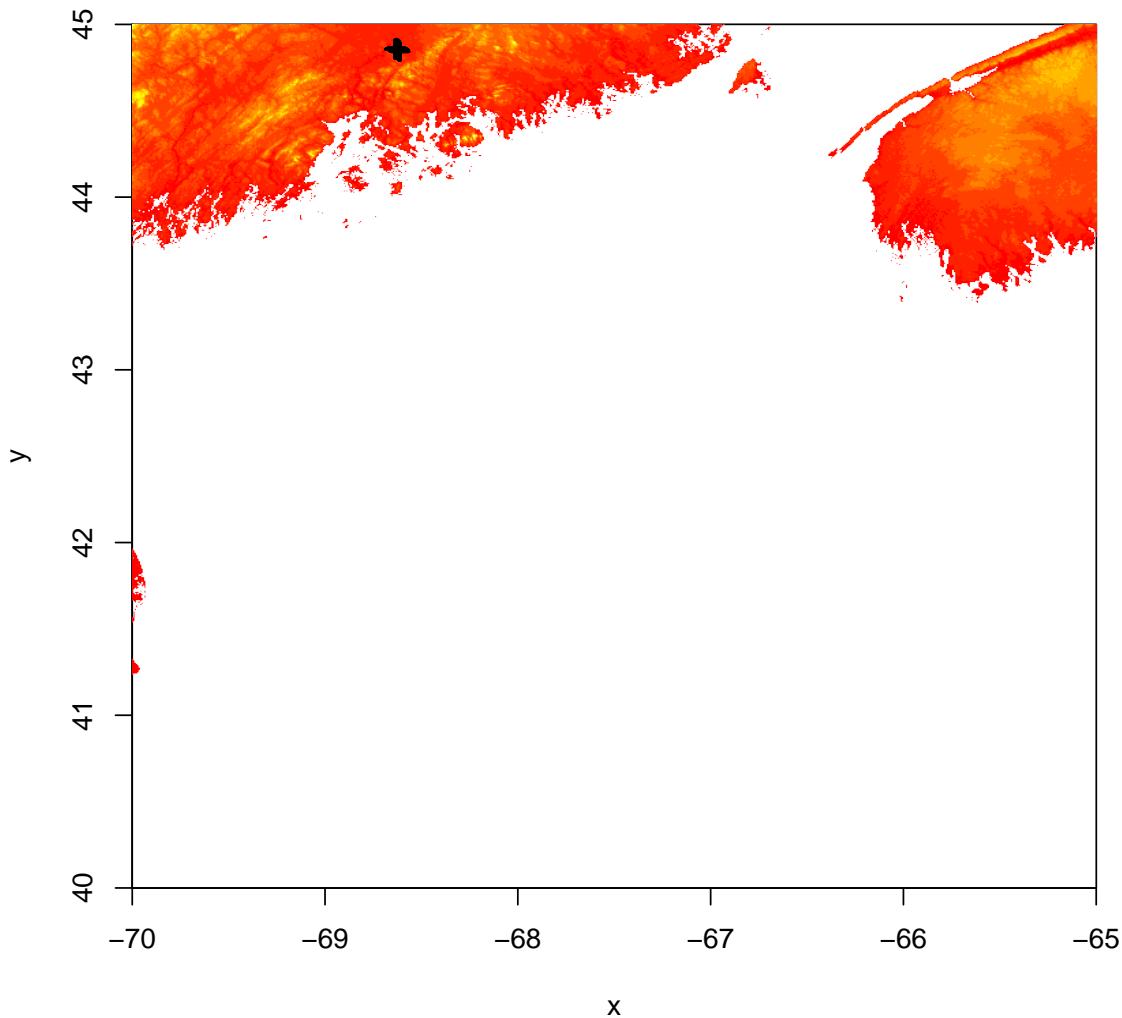
```
> library(raster)
>
> pef.centroid <- as.data.frame(plots) %>%
+   summarize(mu.x = mean(easting), mu.y = mean(northing))
>
> srtm <- getData("SRTM", lon = pef.centroid[1], lat = pef.centroid[2])
>
> class(srtm)

[1] "RasterLayer"
attr(,"package")
[1] "raster"

> proj4string(srtm)

[1] "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"

> image(srtm)
> plot(plots, add = TRUE)
```

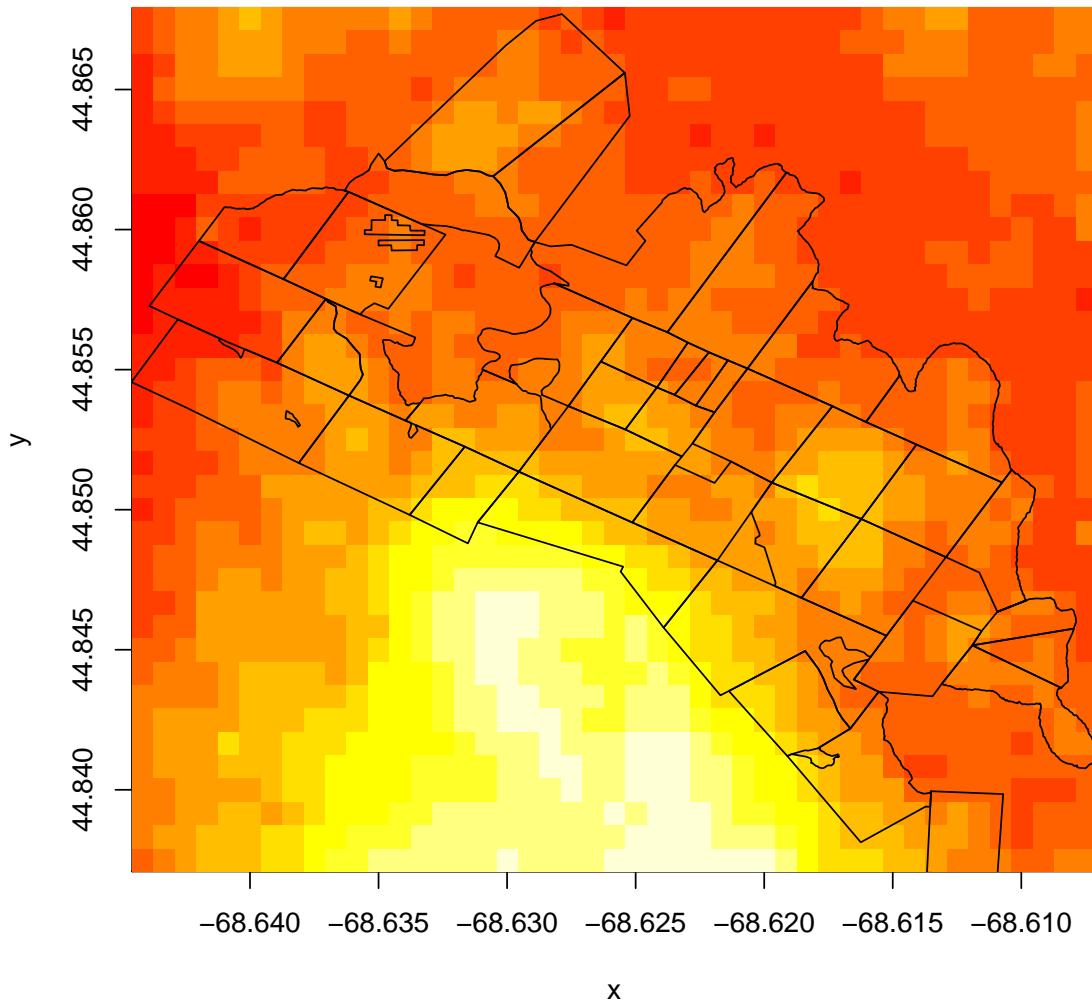


A few things to notice in the code above. First the `getData` function needs the longitude `lon` and latitude `lat` to identify which SRTM raster tile to return (SRTM data come in large raster tiles that cover the globe). As usual, look at the `getData` function documentation for a description of the arguments. To estimate the PEF's centroid coordinate, we averaged the forest inventory plots' latitude and longitude then assigned the result to `pef.centroid`. Second the `srtm` object result from `getData()` is a `RasterLayer`, see Table 11.1. Third `srtm` is in longitude latitude geographic CRS (same as our other PEF data). Finally the image shows SRTM elevation along the coast of Maine, the PEF plots are those tiny specks of black in the northwest quadrant, and the white region of the image is the Atlantic Ocean.

Okay, this is a start, but it would be good to crop the SRTM image to the PEF's extent. This is done using `raster`'s `crop` function. This function can use many different kinds of spatial objects in the second argument to calculate the extent at which to crop

the object in the first argument. Here, I set `mu` as the second argument and save the resulting SRTM subset over the larger tile (the `srtm` object).

```
> srtm <- crop(srtm, mu)
>
> image(srtm)
> plot(mu, add = TRUE)
```



The `crop` is in effect doing a spatial setting of the raster data. We'll return to the `srtm` data and explore different kinds of subsetting in the subsequent sections.

### 11.7.2 Logical, index, and name subsetting

As promised, we can subset spatial objects using the `[]` operator and a logical, index, or name vector. The key is that `sp` objects behave like data frames, see Section 4.5. A logical or index vector to the left of the comma in `[,]` accesses points, lines, polygons, or pixels. Similarly, a logical, index, or name vector to the right of the comma accesses variables.

For example, say we want to map forest inventory plots with more than 10,000 stems per hectare, `plots$stems_ha` (the `min()` was added below to double check that the subset worked correctly).

```
> min(plots$stems_ha)

[1] 119

> plots.10k <- plots[plots$stems_ha > 10000, ]
>
> min(plots.10k$stems_ha)

[1] 10008
```

You can also add new variables to the spatial objects.

```
> plots$diameter_in <- plots$diameter_cm/2.54
>
> head(plots)

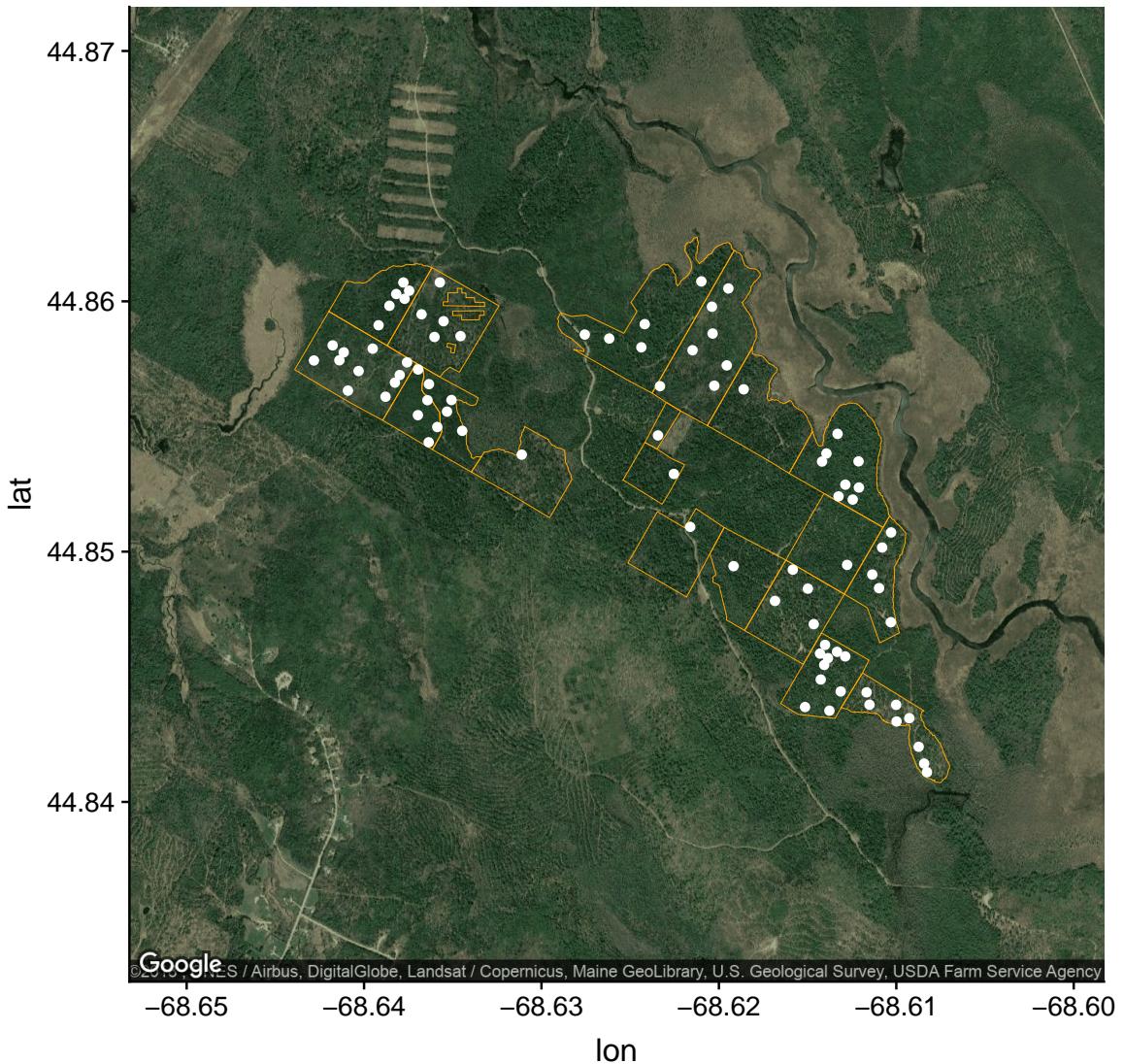
  mu_id plot biomass_mg_ha stems_ha diameter_cm
1   U10    11      96.35     5453       4.8
2   U10    13     115.70     2629       6.9
3   U10    21     121.58     3385       6.1
4   U10    22      71.97     7742       3.1
5   U10    23     122.26     7980       4.7
6   U10    24      85.85    10047       1.6

  basal_area_m2_ha diameter_in
1              22.0  1.8897638
2              23.2  2.7165354
3              23.0  2.4015748
4              16.1  1.2204724
5              29.2  1.8503937
6              19.1  0.6299213
```

### 11.7.3 Spatial subsetting and overlay

A spatial overlay retrieves the indexes or variables from object  $A$  using the location of object  $B$ . With some spatial objects this operation can be done using the `[]` operator. For example, say we want to select and map all management units in `mu`, i.e.,  $A$ , that contain plots with more than 10,000 stems per ha, i.e.,  $B$ .

```
> mu.10k <- mu[plots.10k, ] ## A[B,]
>
> mu.10k.f <- fortify(mu.10k, region = "mu_id")
>
> ggmap(basemap) + geom_polygon(data = mu.10k.f, aes(x = long,
+   y = lat, group = group), fill = "transparent", size = 0.2,
+   color = "orange") + geom_point(data = as.data.frame(plots.10k),
+   aes(x = easting, y = northing), color = "white")
```



More generally, however, the `over` function offers consistent overlay operations for `sp` objects and can return either indexes or variables from object *A* given locations from object *B*, i.e., `over(B, A)` or, equivalently, `B%over%A`. The code below duplicates the result from the preceding example using `over`.

```
> mu.10k <- mu[mu$mu_id %in% unique(over(plots.10k, mu)$mu_id),
+      ]
```

Yes, this requires more code but `over` provides a more flexible and general purpose function for overlays on the variety of `sp` objects. Let's unpack this one-liner into its five steps.

```
> i <- over(plots.10k, mu)
> ii <- i$mu_id
> iii <- unique(ii)
> iv <- mu$mu_id %in% iii
> v <- mu[iv, ]
```

- i* The `over` function returns variables for `mu`'s polygons that coincide with the 85 points in `plots.10k`. No points fall outside the polygons and the polygons do not overlap, so *i* should be a data frame with 85 rows. If polygons did overlap and a point fell within the overlap region, then variables for the coinciding polygons are returned.
- ii* Select the unique `mu` identifier `mu_id` (this step is actually not necessary here because `mu` only has one variable).
- iii* Because some management units contain multiple plots there will be repeat values of `mu_id` in *ii*, so apply the `unique` function to get rid of duplicates.
- iv* Use the `%in%` operator to create a logical vector that identifies which polygons should be in the final map.
- v* Subset `mu` using the logical vector created in *iv*.

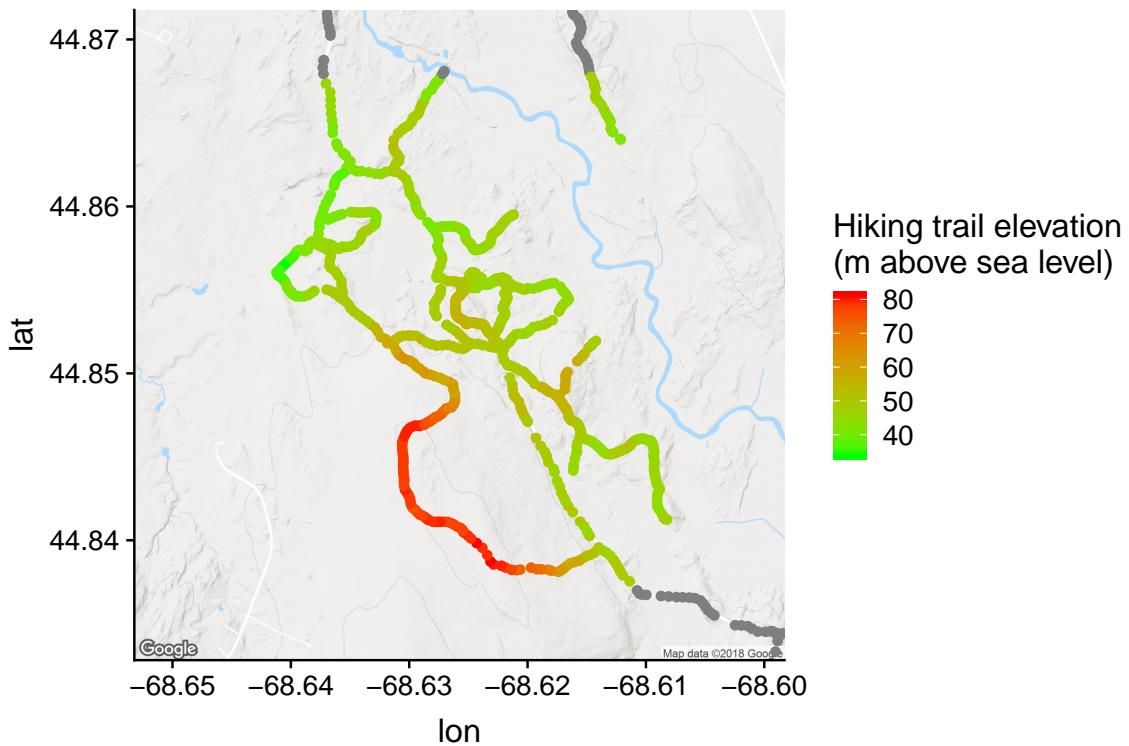
Now let's do something similar using the `srtm` elevation raster. Say we want to map elevation along trails, winter roads, and gravel roads across the PEF. We could subset `srtm` using the `roads` `SpatialLinesDataFrame`; however, mapping the resulting pixel values along the road segments using `ggmap` requires a bit more massaging. So, to simplify things for this example, `roads` is first coerced into a `SpatialPointsDataFrame` called `hikes pts` that is used to extract spatially coinciding `srtm` pixel values which themselves are coerced from `raster`'s `RasterLayer` to `sp`'s `SpatialPixelsDataFrame` called `srtm.sp` so that we can use the `over` function. We also choose a different basemap just for fun.

```
> hikes <- roads[roads$type %in% c("Trail", "Winter", "Gravel"), ]
>
> hikes.pts <- as(hikes, "SpatialPointsDataFrame")
> srtm.sp <- as(srtm, "SpatialPixelsDataFrame")
>
> hikes.pts$srtm <- over(hikes.pts, srtm.sp)
>
> basemap <- get_map(location=mu.bbox, zoom = 14, maptype="terrain")
```

*Warning: bounding box given to google - spatial extent only approximate.  
converting bounding box to center/zoom specification. (experimental)  
Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=44.852323,-68.622222&zoom=14&size=640x480&maptype=terrain&language=en-US&sensor=false*

```
> ggmap(basemap) +
+   geom_point(data=as.data.frame(hikes.pts),
+             aes(x = coords.x1, y = coords.x2, color=srtm[,1])) +
+   scale_color_gradient(low="green", high="red") +
+   labs(color = "Hiking trail elevation\n(m above sea level)",
+        xlab="Longitude", ylab="Latitude")

Warning: Removed 483 rows containing missing values (geom_point).
```



In the call to `geom_point` above, `coords.x1` `coords.x2` are the default names given to longitude and latitude, respectively, when `sp` coerces `hikes` to `hikes.pts`. These points represent the vertices along line segments. The cryptic `.doExtract` warning just means that some vertices points fell outside the `srtm` raster boundaries and were

set to NA (hence the gray points on the map).

Overlay operations involving lines and polygons over polygons require the `rgeos` package which provides an interface to the [Geometry Engine - Open Source](#) (GEOS) C++ library for topology operations on geometries. We'll leave it to you to explore these capabilities.

### 11.7.4 Spatial aggregation

We have seen aggregation operations before when using the `apply` function and `dplyr`'s `summarize` function. The `summarize` function is particularly powerful when combined with `group_by()`, which can apply a function specified in `summarize()` to a variable partitioned using a grouping variable. The `aggregate` function in `sp` works in a similar manner, except groups are delineated by the spatial extent of a thematic variable. In fact, the work we did to create `mu.bio` using `dplyr` functions can be accomplished with `aggregate()`. Using `aggregate()` will, however, require a slightly different approach for joining the derived average `biomass_mg_ha` to the fortified `mu`. This is because the `aggregate` function will apply the user specified function to all variables in the input object, which, in our case, results in an NA for the linking variable `mu_id` as demonstrated below.

```
> mu.ag <- aggregate(plots[, c("mu_id", "biomass_mg_ha")], by = mu,
+   FUN = mean)
>
> head(mu.ag@data, n = 2)

  mu_id biomass_mg_ha
0 <NA>      49.85895
1 <NA>     112.17077
```

With `mu_id` rendered useless, we do not have a variable that uniquely identifies each polygon for use in `fortify`'s `region` argument; hence no way to subsequently join the unfortified and fortified versions of `mu.bio.ag`. Here's the work around. If the `region` is not specified, `fortify()` uses an internal unique polygon ID that is part of the `sp` data object and accessed via `row.names()`<sup>12</sup> So, the trick is to add this unique polygon ID to the `aggregate()` output prior to calling `fortify()` as demonstrated below.

```
> mu.ag$id <- row.names(mu.ag)
>
> mu.ag.f <- fortify(mu.ag)
```

---

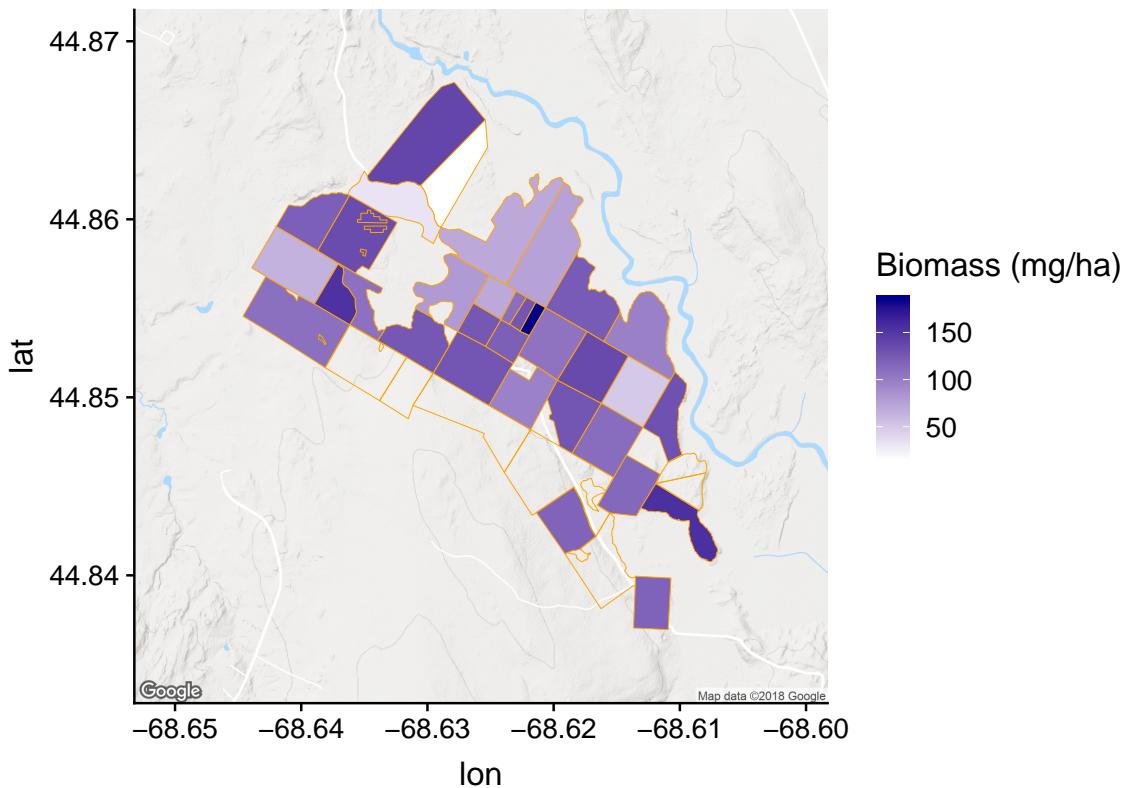
<sup>12</sup>With other data, there is a chance the row names differ from the unique polygon IDs. Therefore a more reliable approach to getting a unique ID is to use `sapply(slot(mu.ag, 'polygons'), function(x) slot(x, 'ID'))`, but replace `mu.ag` with your `SpatialPolygonsDataFrame`. Also, this approach will work with other `sp` objects in right column of Table 11.1.

*Regions defined for each Polygons*

```
> mu.ag.f <- left_join(mu.ag.f, mu.ag@data)

Joining, by = "id"

> ggmap(basemap) + geom_polygon(data = mu.ag.f, aes(x = long, y = lat,
+   group = group, fill = biomass_mg_ha), size = 0.2, color = "orange") +
+   scale_fill_gradient(low = "white", high = "darkblue", na.value = "transparent")
+   labs(fill = "Biomass (mg/ha)")
```



The `aggregate()` function will work with all `sp` objects. For example let's map the variance of pixel values in `srtm.sp` by management unit. Notice that `aggregate()`

is happy to take a user-specified function for FUN.

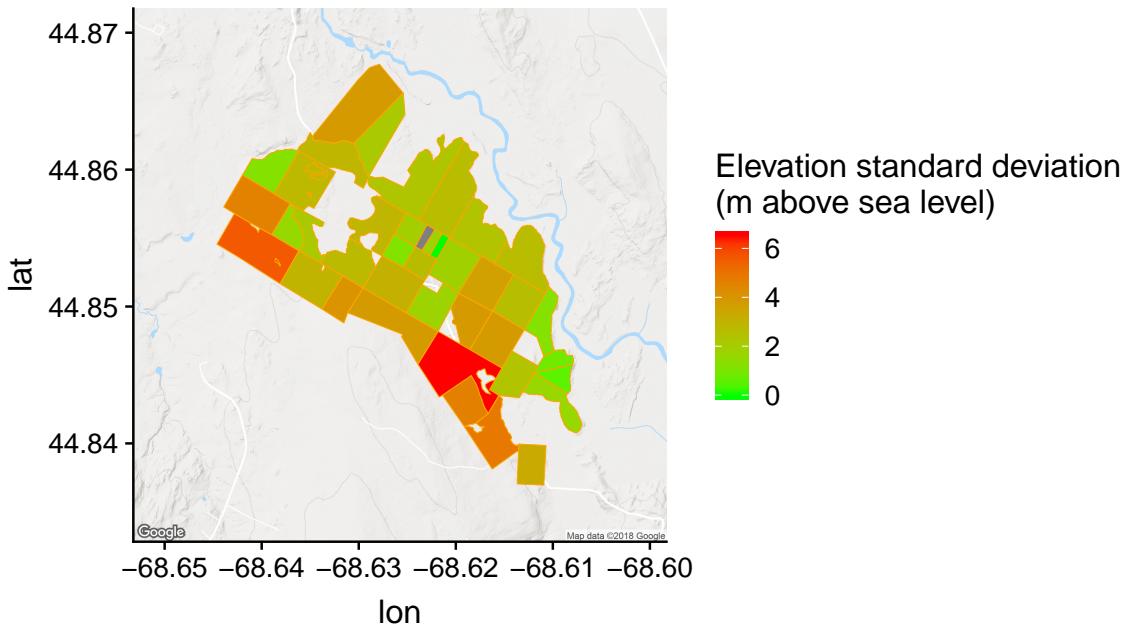
```
> mu.srtm <- aggregate(srtm.sp, by=mu,
+                         FUN=function(x){sqrt(var(x))})
>
> mu.srtm$id <- row.names(mu.srtm)
>
> mu.srtm.f <- fortify(mu.srtm)
```

*Regions defined for each Polygons*

```
> mu.srtm.f <- left_join(mu.srtm.f, mu.srtm@data)
```

*Joining, by = "id"*

```
> ggmap(basemap) +
+     geom_polygon(data=mu.srtm.f, aes(x = long, y = lat, group=group,
+                                         fill=srtm_23_04),
+                   size=0.2, color="orange") +
+     scale_fill_gradient(low="green", high="red") +
+     labs(fill = "Elevation standard deviation\n(m above sea level)",
+          xlab="Longitude", ylab="Latitude")
```



## 11.8 Where to go from here

This chapter just scratches the surface of R's spatial data manipulation and visualization capabilities. The basic ideas we presented here should allow you to take a deeper look into `sp`, `rgdal`, `rgeos`, `ggmap`, `leaflet`, and myriad of other excellent user-contributed R spatial data packages. A good place to start is with Edzer Pebesma's excellent vignette on the use of the map overlay and spatial aggregation, available [here](#), as well as *Applied Spatial Data Analysis with R* by [Bivand et al. \[2013\]](#).

**Exercise 14**

Learning objectives: practice loading and reprojecting spatial data; analyze spatial data; create leaflet maps to convey analysis results; interpret analysis results.

# Chapter 12

## Shiny

Shiny is a framework that turns R code and figures into interactive web applications. Let's start out by looking at a built-in example Shiny app.

```
library(shiny)
runExample("01_hello")
```

In the bottom panel of the Shiny app, we can see the two R scripts that are essential to running any Shiny app: server.R and ui.R. Take a minute to explore how the app works and how the script code is structured. The `server.R` file defines the server logic required to run the app (in this case, to draw the histogram of the Old Faithful Geyser Data). The `ui.R` file defines the user interface (UI) of the app, which affects the layout of the resulting web app.<sup>1</sup>

All of the example code and data for the remainder of this chapter can be accessed using the following code.

```
> library(downloader)
>
> download("http://blue.for.msu.edu/FOR875/data/shiny_chapter_code.zip",
+           destfile = "./shiny_chapter_code.zip", mode = "wb")
>
> unzip("shiny_chapter_code.zip", exdir = ".")
```

### 12.1 Running a Simple Shiny App

Shiny apps allow our users to interact with a data set. Let's say we have access to a dummy data set `name_list.csv`. We begin by just taking a quick look at what this CSV file contains.

---

<sup>1</sup>Since 2014, Shiny has supported single-file applications (one file called `app.R` that contains UI and server components), but in this chapter, we will use the two-file system. See [here](#) for more information.

```

namesDF <- read.csv("http://blue.for.msu.edu/FOR875/data/name_list.csv",
  stringsAsFactors = FALSE)
str(namesDF)

'data.frame': 200 obs. of  4 variables:
 $ First: chr  "Sarah" "Boris" "Jessica" "Diane" ...
 $ Last : chr  "Poole" "Dowd" "Wilkins" "Murray" ...
 $ Age  : int  15 52 12 58 56 4 14 0 28 27 ...
 $ Sex  : chr  "F" "M" "F" "F" ...

```

Our ultimate goal in this chapter will be to explore age distributions for males versus females, but for now, we just show how to setup a Shiny application. Below is the code for simple `server.R` and `ui.R` files. To create and run this Shiny app, follow these steps:

1. Create a new directory called “ShinyPractice.”
2. In the directory, create two blank R scripts: `server.R` and `ui.R`. Also, copy the file, `name_list.csv` into the “ShinyPractice” directory.
3. Copy the code below into the respective scripts.
4. Run the Shiny app from RStudio. There are two ways to do this: 1) use the RStudio button (See Figure 12.1) or; 2) type the function `runApp()` in the RStudio console.

If everything compiles correctly, you should get a window that looks like Figure 12.2. There are a few things to notice. First, our Shiny app is not interactive yet because there is no way for a user to input any data—the page is view-only. Second, in our `server.R` file, our function, `function(input,output) {}`, reflecting that we are not yet using any user inputs or creating any outputs. Last, note that we read in the CSV file in the `server.R` script.

### **ui.R version 1**

```

# ui.R

library(shiny)

fluidPage(
  titlePanel("Random Names Analysis"),

```

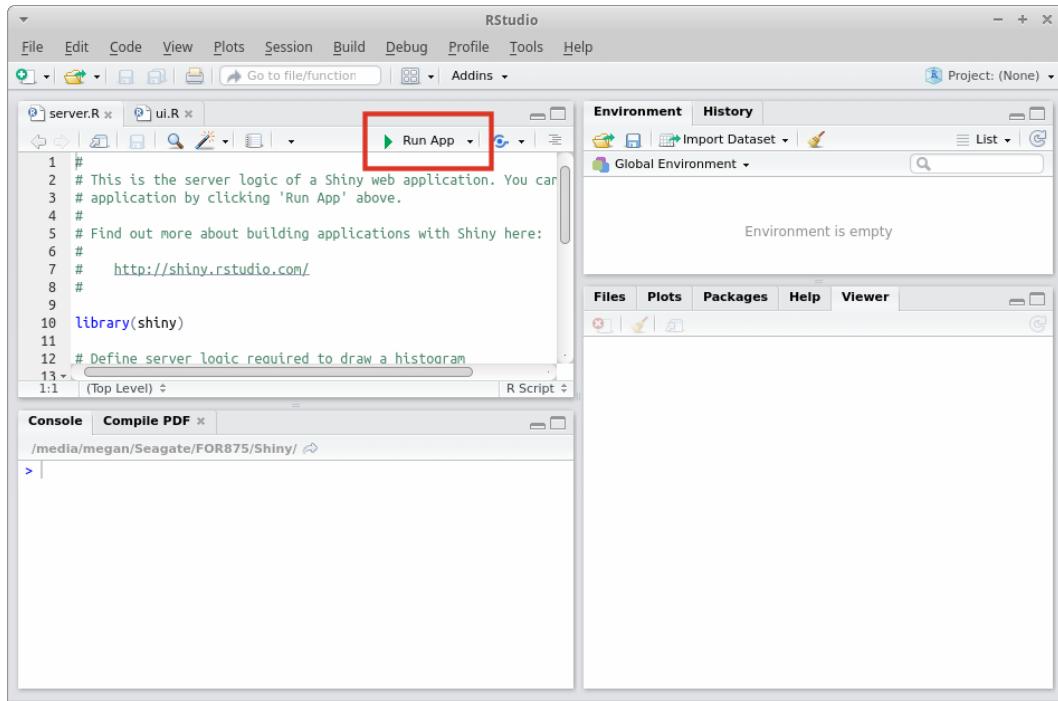


Figure 12.1: running a Shiny app locally.

```
sidebarLayout(  
  
  sidebarPanel("our inputs will be here"),  
  
  mainPanel("our output will appear here")  
)  
)
```

### server.R version 1

```
# server.R  
  
library(shiny)  
  
names.df <- read.csv("http://blue.for.msu.edu/FOR875/data/name_list.csv")  
  
function(input, output) {  
}
```

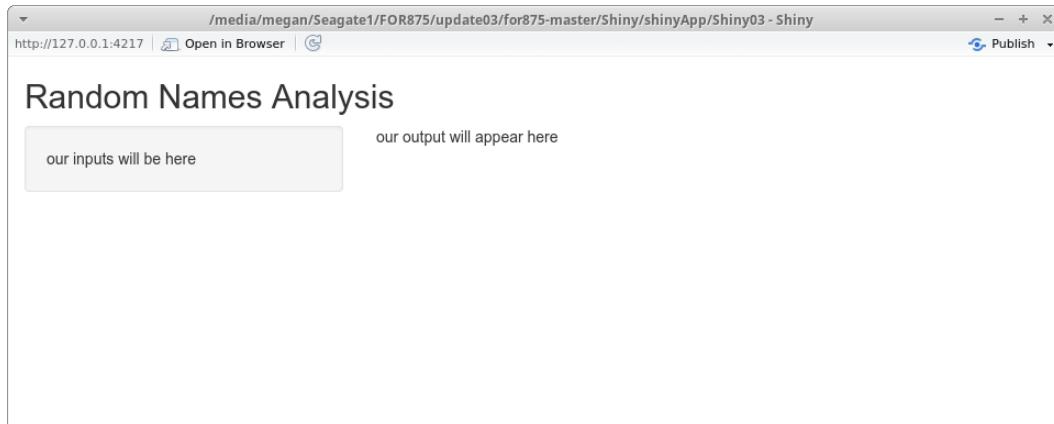


Figure 12.2: Your first Shiny app.

## 12.2 Adding User Input

In order to have user input, we must include widgets. There are many functions in the Shiny package that create these interactive inputs. Types of input that we will use in this chapter include slider bars, dropdown menus, and check boxes. Each of these types of functions correspond to a function that will always take in at minimum an `inputId` and a `label` as arguments. The `inputId` argument is the name that will be used to access the value that the user selects. (We will refer to this when we update the `server.R` function.) Our users will not see this name. The `label` argument, however, is what the user sees above the data input area.

We begin by adding a dropdown menu to our `ui.R` script with the `selectInput` function. Update `ui.R` and rerun the application. What is our `inputId` for the dropdown menu? What is our `label`? Can you see where the `label` is displayed in the app when you rerun it?

Look at the structure of the `choices` argument in `selectInput`. It is a vector of options in the form "name" = "value" where the `name` is what the user sees. The `value` is what is returned to the `server.R` file when the user selects an option.

### ui.R version 2

```
# ui.R

library(shiny)

fluidPage(
  titlePanel("Random Names Age Analysis"),
  sidebarLayout(
    sidebarPanel(
```

```

# Dropdown Selection for Male/Female
selectInput("sexInput", "Sex:",
            choices = c("Female" = "F",
                        "Male" = "M",
                        "Both" = c("M", "F")))
),
mainPanel("our output will appear here")
)
)

```

## 12.3 Adding Output

Now that we've formatted the user input part of our application in `ui.R`, let's make the app truly interactive by changing the output depending on the user input. Now is the time to modify the `server.R` script. Our goal is to plot a histogram in the main panel. The histogram should only plot the age distribution of the sex selected by the user.

### 12.3.1 Interactive Server Logic

Server logic is defined by two arguments: `input` and `output`. These objects are both list-like, so they can contain multiple other objects. We already know from our `ui.R` file that we have an input component called `sex`, which we can access in the reactive portion of `server.R` by calling `input$sex`. We then will create a plot object called `histogram` that we can modify in `server.R` by calling `output$histogram`.

Reactive portions of our `server.R` script are inside the `server` function. We create reactive outputs using Shiny's functions like `renderPlot`.<sup>2</sup> Obviously, `renderPlot` renders the contents of the function into a plot object; this is then stored in the `output` list wherever we assign it (in this case, `output$histogram`). Notice that the contents of the `renderPlot` function are contained not only by regular parentheses, but also by curly brackets.

In this server function, we simply subset the data frame by the user's chosen sex and create the histogram of ages for that subset. Since the `hist` function is the last plot-type function called in `renderPlot`, the resulting histogram is stored as `output$histogram`.

### server.R version 2

---

<sup>2</sup>Every reactive output function's name in Shiny is of the form “`render*`”.

```
# server.R

library(shiny)

# read in the data frame
names.df <- read.csv("http://blue.for.msu.edu/FOR875/data/name_list.csv")

# Define server logic required to draw a histogram
function(input, output) {

  output$histogram <- renderPlot({

    if (input$sex != "B") {
      subset.names.df <- subset(names.df, Sex == input$sex)
    } else {
      subset.names.df <- names.df
    }

    ages <- subset.names.df$Age
    # draw the histogram with the specified 20 bins
    hist(ages, col = "darkgray", border = "white")
  })
}
```

Update your `server.R` function to match the code above. Rerun the application. Note that the appearance of the app doesn't change because we have not updated `ui.R`.

### 12.3.2 Interactive User Interface

Now we update `ui.R` to make the app interactive. The `plotOutput("histogram")` function access the `histogram` component of the `output` list and plots it in the main panel. Copy the code below and rerun the application. You have now created your first Shiny app!

#### **ui.R version 3**

```
# ui.R

library(shiny)
```

```

fluidPage(
  titlePanel("Random Names Age Analysis"),
  sidebarLayout(
    sidebarPanel(
      # Dropdown Selection for Male/Female
      selectInput("sexInput", "Sex:",
                  choices = c("Female" = "F",
                              "Male" = "M",
                              "Both" = c("M", "F"))))
    ),
    mainPanel(plotOutput("histogram"))
  )
)

```

Visit the <https://shiny.rstudio.com/gallery/widget-gallery.html> for example code for all types of Shiny Widgets.

## 12.4 More Complicated Shiny App: Michigan Campgrounds

The Michigan DNR website has multiple open-source data files available to download in a variety of formats. The Michigan State Park Campgrounds data is available [here](#) to download as a CSV file. Let's view this data and see what we can do with it.

```

> u <- "http://blue.for.msu.edu/FOR875/data/Michigan_State_Park_Campgrounds.csv"
> sites <- read.csv(u, stringsAsFactors = F)
> str(sites)

'data.frame': 151 obs. of 19 variables:
 $ X       : num -84.4 -84.4 -83.8 -83.8 -83.8 ...
 $ Y       : num 42.9 42.9 42.5 42.5 42.5 ...
 $ OBJECTID: int 10330 10331 5986 5987 5988 5989 5990 5991 5992 5993 ...
 $ Type    : chr "Campground" "Campground" "Campground" "Campground" ...
 $ Detail_Type: chr "State Park" "State Park" "State Park" "State Park" ...
 $ DISTRICT: chr "Rose Lake" "Rose Lake" "Rose Lake" "Rose Lake" ...
 $ COUNTY  : chr "Clinton" "Clinton" "Livingston" "Livingston" ...
 $ FACILITY: chr "Sleepy Hollow State Park" "Sleepy Hollow State Park" "Brighton ...
 $ Camp_type: chr "Modern" "Organizational" "Modern" "Organizational" ...
 $ TOTAL_SITE: int 203 1 144 10 25 25 219 75 2 297 ...
 $ ADA_SITES: int 0 0 0 0 0 0 3 0 13 ...
 $ name    : chr "Sleepy Hollow State Park" "Sleepy Hollow State Park" "Brighton ...
 $ Ownership: chr "State" "State" "State" "State" ...

```

Table 12.1: Interactive elements

| Element ID      | Description                        | Function to Create |
|-----------------|------------------------------------|--------------------|
| input\$rangeNum | desired range of campsite quantity | sliderInput        |
| input\$type     | desired campsite type              | selectInput        |
| input\$ada      | desired ADA site availability      | checkboxInput      |
| output\$plot1   | map with campground markers        | renderPlot         |
| output\$text1   | HTML-formatted list of campgrounds | renderText         |

```
$ Management : chr  "DNR" "DNR" "DNR" "DNR" ...
$ Surface    : chr  "Dirt" "Dirt" "Dirt" "Dirt" ...
$ Use_       : chr  "Recreation" "Recreation" "Recreation" "Recreation" ...
$ Condition  : chr  "Unknown" "Unknown" "Unknown" "Unknown" ...
$ Lat        : num   42.9 42.9 42.5 42.5 42.5 ...
$ Long       : num  -84.4 -84.4 -83.8 -83.8 -83.8 ...
```

We see that Michigan has 151 state park campgrounds, and our data frame contains 19 variables. Let's create a Shiny app UI in which the user selects desired campground specifications, and the app displays the map of campgrounds and the list of campgrounds. In the example code in the `CampsitesMI` directory, we have our two Shiny app files: `ui.R` and `server.R`. Start out by running the application to see how it works. Examine the example code and how it relates to the application that you are running. For a summary of the input and output elements, see Table 12.1.

### 12.4.1 Michigan Campgrounds UI

First, let's look at the structure of the page. Like before, we have a `fluidPage` layout with a title panel and sidebar. The sidebar contains a sidebar panel and a main panel. Our sidebar panel has three user input widgets:

- `sliderInput`: Allows user to specify a range of campsites desired in their campground. Since the maximum number of campsites in any Michigan state park campground is 411, 420 was chosen as the maximum.
- `selectInput`: Allows user to select what type of campsites they want. To get the entire list of camp types, we used the data frame, `sites`, loaded at the beginning of the script.
- `checkboxInput`: Allows the user to see only campgrounds with ADA sites available.

### 12.4.2 Michigan Campgrounds Server Logic

In our `server.R` script, we have two functions that fill in our output elements:

- `renderText`: Creates a character string of HTML code to print the bulleted list of available sites.
- `renderPlot`: Creates a `ggplot2` map of Michigan with campground markers.

Note that both of these functions contain identical subsetting of the `sites` data frame into the smaller `sites1` data frame (see below). As you can see from the code, we use the three inputs from the application to subset the data: `rangeNum` from the slider widget, `type` from the dropdown menu, and `ada` from the checkbox.

This repeated code can be avoided using Shiny's reactive expressions. These expressions will update in value whenever the widget input values change. See [here](#) for more information. The use of reactive expressions is beyond the scope of this chapter, but it is an important concept to be familiar with if you plan to regularly create Shiny applications.

```
# lines 14 and 38 in server.R
sites1 <- subset(sites,
                  TOTAL_SITE >= input$rangeNum[1] & //
                  TOTAL_SITE <= input$rangeNum[2] & //
                  Camp_type == input$type & //
                  if(input$ada){ ADA_SITES > 0 } else {ADA_SITES >= 0}
                )
```

## 12.5 Adding Leaflet to Shiny

[Leaflet](#) can be easily incorporated in Shiny apps. In the Michigan Campgrounds example code, we used `plotOutput` and `renderPlot` to put a plot widget in our Shiny app. Similarly, in this code, we will use `leafletOutput` and `renderLeaflet` to add Leaflet widgets to our app. Run the Shiny app in the `CampsitesMI_Leaflet` directory. What are the differences between the plot widget in this app and the plot widget in the previous app (using `ggplot2`)?

Our code inside `renderLeaflet` is displayed below. As a reminder from our previous use of Leaflet, we can use the `magrittr` package's pipe operator, `%>%`, to add properties to our Leaflet plot. In the `addCircleMarkers` function, you can see that we used the `mapply` function to apply HTML code to each of the markers. The HTML code simply prints the site name and number of sites within each marker label. The `else` statement serves the purpose that if our subset, `sites1`, is empty, we render a map centered on Lansing, MI.

```
# renderLeaflet function from server.R
output$plot1 <- renderLeaflet({
```

```

# create a subset of campsites based on inputs
sites1 <- subset(sites,
                  TOTAL_SITE >= input$rangeNum[1] & //
                  TOTAL_SITE <= input$rangeNum[2] & //
                  Camp_type == input$type & //
                  if(input$ada){ ADA_SITES > 0 } else {ADA_SITES >= 0})

if(nrow(sites1) > 0){
  leaflet(sites1) %>% addTiles() %>%
    addCircleMarkers(lng = ~Long, lat = ~Lat,
                     radius = 5,
                     color = "red",
                     label = mapply(function(x,y) {
                       HTML(sprintf('<em>%s</em><br>%s site(s)',

                                     htmlEscape(x),
                                     htmlEscape(y))),
                     sites1$FACILITY, sites1$TOTAL_SITE
                     )
    )
} else {
  leaflet() %>% addTiles() %>%
  setView( -84.5555, 42.7325, zoom = 7)
}

})

```

## 12.6 Why use Shiny?

In this chapter, we learned what a Shiny app is, what it's components are, how to run the app, and how to incorporate Leaflet plots. We can also host these apps online on our own server or the [shinyapps.io](#) server, which can be accessed directly from RStudio. Hosting our apps on a server allows anyone with internet access to interact with our widgets.

The [shinyapps.io](#) server allows free hosting of apps within the monthly limits of 5 applications and 25 active hours. Paid plans are also available. A user guide to deploying your application on shinyapps.io is available [here](#). Set up a free account and impress your friends by sending them links to your Shiny apps! In addition to hosting your applications, here are a few more things to discover with Shiny:

- [Learn](#) how to use reactive values.
- [Incorporate](#) CSS stylesheets to make your apps fancier.

- [Explore](#) the single-file Shiny app structure.

### Exercise 15

Learning objectives: practice updating `ggplot2` plot aesthetics; modify Shiny HTML output; add an interactive Shiny element.

# Chapter 13

## Rcpp

R is an interpreted language. The code interpreter runs each line of a script individually instead of compiling the entire file at once as in a compiled language like C++. This means R can be slower than compiled languages when completing some tasks. In this chapter, we show how to improve the performance of our R scripts by rewriting functions in C++.<sup>1</sup> Although R has an API for developers to move between R data structures and data structures in lower-level languages (C++, C, FORTRAN, etc.), an alternative is to use `Rcpp`. According to the [official Rcpp website](#), “Rcpp provides a powerful API on top of R, permitting direct interchange of rich R objects between R and C++.” It is a higher-level approach than using the aforementioned API directly. C++ functions can address gaps in R’s abilities including

- Vectorizing loops whose subsequent iterations depend on previous iterations.
- Writing recursive functions with lower overhead. (Recursive functions are functions that call themselves, leading to millions of function calls.)
- Using data structures and algorithms that R does not provide but that are in the C++ standard template library (STL) such as ordered maps and double-ended queues.

In this chapter, we will cover an abbreviated/modified version of Hadley Wickham’s tutorial “[High performance functions with Rcpp](#).”

### 13.1 Getting Started with Rcpp

#### 13.1.1 Installation

Before we install the `Rcpp` package, we need a working C++ compiler. To get the compiler:

---

<sup>1</sup>You do not need prior C++ experience to complete this chapter and corresponding exercises, but we are hoping you learn a little C++ syntax along the way!

- Windows: install Rtools from [here](#).
- Mac: install Xcode from the Mac App Store.
- Linux: in the terminal (for Debian-based systems or similar) enter  
`sudo apt-get install r-base dev`.

Now we can install Rcpp. (Note that the `microbenchmark` package is needed for the examples in this chapter, so we'll install that, too.)

```
> install.packages(c("Rcpp", "microbenchmark"))
```

### 13.1.2 The Simplest C++ Example

It is common for textbooks to introduce a new programming language through the “Hello World” program.<sup>2</sup> If we wanted to write this simple program in C++, the code would look like this:

```
#include <iostream>

int main()
{
    std::cout << "HelloWorld!" << std::endl;
}
```

Think about the C++ code above. How would you accomplish the same task in R? How are the programs similar in syntax and structure? How are they different? In the following section, we will rewrite this program in a format that can interface with R through Rcpp. Note that C++ is a compiled language, which means we cannot run the program line-by-line like we can with an R script.

## 13.2 Using Rcpp

### 13.2.1 Exporting C++ Functions

Let's start out with that simple, obligatory “Hello World!” program in C++ that we will export to R. C++ files use the extension `*.cpp`. Through the RStudio menu, create a new C++ File (File > New File > C++ File). Note that the default C++ file contains the line `#include <Rcpp.h>`. In C++, these `include` files are header files that provide us with objects and functions that we can use within our C++ file. Although the program building process is different, there are some parallels between

---

<sup>2</sup>For a collection of “Hello World” programs in 400+ programming languages, see [here](#).

these header files and the packages we use in our R code. Note that unlike R, every line of code in a C++ file must end with a semicolon.

To verify that Rcpp is working correctly, let's run the following code. Save the document as `hello.cpp`, and enter the following code:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]

void hello(){
  Rprintf("HelloWorld!\n");
}
```

Let's do a quick line-by line explanation:

1. `#include <Rcpp.h>`: contains the definitions used by the Rcpp package.
2. `using namespace Rcpp;`: allows us to call functions from the Rcpp header file by writing `functionname` instead of `Rcpp::functionname`.
3. `// [[Rcpp::export]]`: an Rcpp attribute. Attributes are annotations that are added to C++ source files to indicate that C++ functions should be made available as R functions.
4. `void hello(){}`: the declaration of the `hello` function, which has a return type `void` and no arguments, hence the empty parentheses. Note that the function content (definition) is all contained within the squiggly brackets.
5. `Rprintf("Hello World!\n");`: prints to the R console.

Now, we can go to the RStudio Console and test our code with the following three lines and the expected output.

```
> library(Rcpp)
> sourceCpp("hello.cpp")
> hello()
```

```
[1] "Hello World!"
```

The `sourceCpp` function parsed our C++ file (“`hello.cpp`”) and looked for functions marked with the `Rcpp::export` attribute. A shared library was then built and the exported function (`hello()`) was made available as an R function in the current environment. The function simply printed “Hello World!” to our RStudio Console. Great! You have now written a C++ function using built-in C++ types and Rcpp wrapper types and then sourced them like an R script.

### 13.2.2 Inline C++ Code

Maintaining C++ code in its own source file provides several benefits including the ability to use C++ aware text-editing tools and straightforward mapping of compilation errors to lines in the source file. However, it's also possible to do inline declaration and execution of C++ code.

There are several ways to accomplish this, including passing a code string to `sourceCpp()` or using the shorter-form `cppFunction()` or `evalCpp()` functions. Run the following code:

```
library(Rcpp)

cppFunction("int add(int x, int y, int z){
  int sum = x + y + z;
  return sum;
}")

add(1, 2, 4)

[1] 7

a <- add(3, 2, 1)
a

[1] 6
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. We're going to use this simple interface to learn how to write C++. C++ is a large language, and there's no way to cover it all in the limited time we have. Instead, you'll get the basics so that you can start writing useful functions to address bottlenecks in your R code.

## 13.3 The Rcpp Interface

The following subsections will teach you the basics by translating simple R functions to their C++ equivalents. We'll start simple with a function that has no inputs and a scalar output, and then get progressively more complicated:

- No input and scalar output
- Scalar input and scalar output
- Vector input and scalar output

- Vector input and vector output
- Matrix input and vector output
- Matrix input and matrix output

### 13.3.1 No input and scalar output

Let's start with a very simple function. It has no arguments and always returns the integer 1:

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
    return 1;
}
```

We can compile and use this from R with `cppFunction`:

```
cppFunction("int one() {
    return 1;
}")
```

This small function illustrates a number of important differences between R and C++:

- The syntax to create a function looks like the syntax to call a function; you don't use assignment to create functions as you do in R.
- You must declare the type of output the function returns. This function returns an int (a scalar integer). The classes for the most common types of R vectors are: NumericVector, IntegerVector, CharacterVector, and LogicalVector.
- Scalars and vectors are different. The scalar equivalents of numeric, integer, character, and logical vectors are: double, int, String, and bool.
- You must use an explicit return statement to return a value from a function.
- Every statement is terminated by a semicolon.

### 13.3.2 Scalar input and scalar output

The next example function implements a scalar version of the `sign()` function which returns 1 if the input is positive, and -1 if it's negative:

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction("int signC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}")
```

In the C++ version:

- We declare the type of each input in the same way we declare the type of the output. While this makes the code a little more verbose, it also makes it very obvious what type of input the function needs.
- The `if` syntax is identical — while there are some big differences between R and C++, there are also lots of similarities! C++ also has a `while` statement that works the same way as R's, e.g., as in R you can use `break` to exit the loop.

### 13.3.3 Vector input and scalar output

One big difference between R and C++ is that the cost of loops is much lower in C++. For example, we could implement the `sum()` function in R using a loop. If you've been programming in R a while, you'll probably have a visceral reaction to this function!

```
sumR <- function(x) {
  total <- 0
  for (i in 1:length(x)) {
```

```

        total <- total + x[i]
    }
    total
}

```

In C++, loops have very little overhead, so it's fine to use them.

```

cppFunction("double sumC(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; i++) {
        total += x[i];
    }
    return total;
}")

```

The C++ version is similar, but

- To find the length of the vector, we use the `.size()` method, which returns an integer. C++ methods are called with `.` (i.e., a period).
- The `for` statement has a different syntax: `for(init; check; increment)`. This loop is initialized by creating a new variable called `i` with value 0. Before each iteration, we check that `i < n`, and terminate the loop if it's not. After each iteration, we increment the value of `i` by one, using the special prefix operator `++` which increases the value of `i` by 1.
- In C++, vector indices start at 0. I'll say this again because it's so important: IN C++, VECTOR INDICES START AT 0! This is a very common source of bugs when converting R functions to C++.
- Use `=` for assignment, not `<-`.
- C++ provides operators that modify in-place: `total = x[i] +` is equivalent to `total = total x[i] +`. Similar in-place operators are `-=`, `*=`, and `/=`.

This is a good example of where C++ is much more efficient than R. As shown by the following microbenchmark, `sumC()` is competitive with the built-in (and highly optimized) `sum()`, while `sumR()` is several orders of magnitude slower. Note, you'll need to install the `microbenchmark` package.

```

library(microbenchmark)

x <- runif(1000)
microbenchmark(sum(x), sumC(x), sumR(x))

```

```
Unit: nanoseconds
```

|         | expr  | min   | lq       | mean  | median | uq      | max | neval |
|---------|-------|-------|----------|-------|--------|---------|-----|-------|
| sum(x)  | 821   | 822   | 1055.66  | 822   | 1232   | 4927    | 100 |       |
| sumC(x) | 2053  | 2053  | 15400.19 | 2054  | 2464   | 1227128 | 100 |       |
| sumR(x) | 55424 | 55424 | 96943.20 | 55425 | 55835  | 4151873 | 100 |       |

The `microbenchmark()` function runs each function it's passed 100 times and provides summary statistics for the multiple execution times. This is a very handy tool for testing various function implementations (as illustrated above).

### 13.3.4 Vector input and vector output

Next we'll create a function that computes the Euclidean distance between a value and a vector of values:

```
pdistR <- function(x, ys) {
  sqrt((x - ys)^2)
}
```

It's not obvious that we want `x` to be a scalar from the function definition. We'd need to make that clear in the documentation. That's not a problem in the C++ version because we have to be explicit about types:

```
cppFunction("NumericVector pdistC(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);

  for(int i = 0; i < n; i++) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}")
```

This function introduces only a few new concepts:

- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n)`. This and similar vector and matrix constructors initialize the elements with zeros. Another useful way of making a vector is to copy an existing one: `NumericVector zs = clone(ys)`.
- C++ uses `pow()`, not `^`, for exponentiation.

Note that because the R version is fully vectorized, it's already going to be fast. On my computer, it takes around 8 ms with a 1 million element `y` vector. The C++

function is twice as fast,  $\sim 4$  ms, but assuming it took you 10 minutes to write the C++ function, you'd need to run it  $\sim 150,000$  times to make rewriting worthwhile :-) The reason why the C++ function is faster is subtle, and relates to memory management. The R version needs to create an intermediate vector the same length as  $y$  ( $x - ys$ ), and allocating memory is an expensive operation. The C++ function avoids this overhead because it uses an intermediate scalar.

### 13.3.5 Matrix input and vector output

Each vector type has a matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix`, and `LogicalMatrix`. Using them is straightforward. For example, we could create a function that reproduces `rowSums()`:

```
cppFunction("NumericVector rowSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}")

x <- matrix(sample(100), 10)

rowSums(x)

[1] 410 481 531 549 533 488 521 329 607 601

rowSumsC(x)

[1] 410 481 531 549 533 488 521 329 607 601
```

Let's look at the main differences:

- In C++, you subset a matrix with `()`, not `[]`.
- In C++, use `.nrow()` and `.ncol()` methods to get the dimensions of a matrix.

### 13.3.6 Matrix input and matrix output

See the Rcpp exercise.

#### Exercise 16

Learning objectives: practice using Rcpp to run a C++ function through R; use `microbenchmark()` to compare function performance.

# Bibliography

Roger S. Bivand, Edzer Pebesma, and Virgilio Gomez-Rubio. *Applied spatial data analysis with R, Second edition*. Springer, NY, 2013.

Jenny Bryan. Stat 545 topics. <http://stat545-ubc.github.io/topics.html>. Accessed: 2015-08-02.

Winston Chang. *R Graphics Cookbook*. O'Reilly Media, Inc., 2013. ISBN 1449316956, 9781449316952.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009. URL <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.

G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York, 2014. ISBN 9781461471370. URL <https://books.google.com/books?id=at1bmAEACAAJ>.

Phil Spector. *Data Manipulation With R*. Use R! 2008. ISBN 0-387-74730-3.

Wegmann. Raster vector tikz.png, 2010. URL [https://commons.wikimedia.org/wiki/File:Raster\\_vector\\_tikz.png](https://commons.wikimedia.org/wiki/File:Raster_vector_tikz.png). File: Raster\_vector\_tikz.png.

Hadley Wickham. ggplot2 documentation. <http://docs.ggplot2.org>. Accessed: 2015-08-02.

Hadley Wickham and Carson Sievert. *ggplot2: Elegant Graphics for Data Analysis*. Springer, Cham, 2nd edition, 2016. ISBN 9783319242750;331924275X.